WHOI-90-44

The Seadata Program

by

Thomas W. Danforth

Woods Hole Oceanographic Institution
Woods Hole, Massachusetts 02543

October, 1990

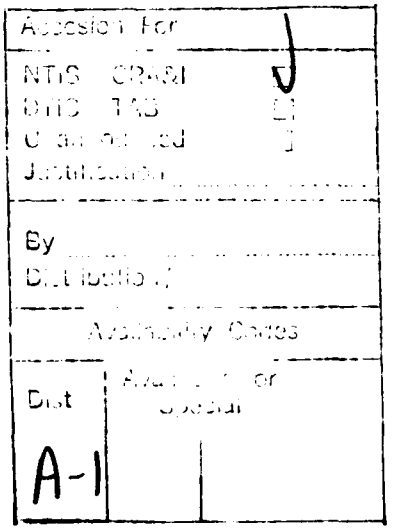
Technical Report

Funding was provided by the Office of Naval Research through Contract No. N00014-84-C-0134 and the U.S. Geological Survey under Contract No. 14-08-0001-A0245.

Approved for Distribution:

James R. Luyten, Chairman
Department of Physical Oceanography

Table of Contents



Abstract

Current meter and meteorological instrument data are typically stored in the instrument on cassette tapes. Seadata, described in this report, is a PC version of the original CARP program (CAssette Reading Program) which transferred the data and prepared it for further processing. Also described are two programs which provide byte swapping which is necessary to use the PC data on a VAX/VMS computer. Some changes to the CARP format have been made and are documented here.

The Seadata Program

Introduction

Seadata is a personal computer (PC) version of the CARP program (CAsette Reading Program) which was written by Mary Hunt in 1972 for the Hewlett Packard (HP) 2100 series of computers. CARP was written to read current meter data which was stored on cassette tapes in the instruments. It was rewritten by Jerry Needell in 1983 as CRAP for use on LSI-11 computers. Both CARP and CRAP read data sent from a Model 12 cassette tape reader manufactured by Sea Data Corp., Newton, Mass. This version builds on the earlier versions by taking advantage of PC technology and the capabilities of the MS-DOS operating system.

In the process of writing seadata, the original CARP and CRAP data formats have been extended. As a result, two other programs (PCARPHP and PCARP) have been written to make the output from seadata look as though it came from an HP or an LSI computer. These programs run on a VAX running VMS and are also documented here.

While writing these programs, I have had the assistance of a number of people. Among them are Ken Prada, Melora Park Samelson, Robin Singer and Dave Aubrey.

Operating environments

Seadata is written in Turbo C and will run on 80286 or 80386 based PC's running MS-DOS Version 3.2 or greater. It also requires a parallel interface card based on an Intel 8255 processor which, in this case, was purchased from Optimal Technology, Inc. (part IB-24). This interface was modified so it would be compatible with the signals sent from the Model 12 tape reader. The modifications to the

interface and the cabling to connect it to the PC are documented in Appendices 1 and 2.

Since most of the programs which process cassette data are run on a VAX/VMS computer, the two programs (PCARP and PCARPHP) which convert the output to the CRAP or CARP format are written in VAX C and run under VMS.

Overview of the data flow

In order for the current meter data recorded on cassette tapes to be processed on a VAX running VMS, the data must go through several steps to get it ready. The first step is reading the tape using a Sea Data Model 12 reader. The tape reader is connected to a PC through a special parallel interface and the seadata program is run to collect the data and store it on the PC's hard disk. The output from the seadata program is two files with the extensions ".CMM" and ".DAT" for comment and data respectively. The comment file (".CMM") is a printable ascii file which contains some header information, the comments entered by the user, and information about the number of records processed and errors encountered. The data file (".DAT") is a modification of the CARP data files and contains the binary data read from the tape.

The ".CMM" and ".DAT" files can be used for processing on the PC if desired or they can be copied to a VAX for processing under the existing Buoy processing system. The files can be copied to a VAX in one of two ways. The first uses the file transfer program, ftp, and sends the files over ethernet. The ethernet transfer must be done in ascii mode for the ".CMM" file and binary mode for the ".DAT" file. An example of an ethernet file transfer is documented in Appendix 3, also documentation on ftp may be useful.

The second method of file transfer is via 9 track, ANSI labeled tape. ANSI tapes which are not created on VAX/VMS do not have some of the file handling information in the file header which is used by VMS. If tape copies are used, the

number of bytes in each record on the tape will be different than when the file is copied by ethernet. The files copied via ANSI labeled tape have an extra byte of carriage control which must be eliminated.

Since the tapes read on a PC are most likely to be processed on a VAX, two programs were written to convert the PC's data format to the VAX's data format. The program to produce a CRAP data file (looking like it came from an LSI) is called PCARP. This program requires the input of the two data files (".CMM" and ".DAT") and will produce a third file with an extension of ".LSI". The ".LSI" file is acceptable as input to the CARPBIT program used as part of the Buoy processing system. The program to produce a CARP format data file (looking like it came from an HP) is called PCARPHP. It also requires the input of the two data files and will produce a third file with an extension of ".HP". The ".HP" file can be used in processing by the USGS. PCARPHP also removes the extra carriage control characters introduced by copying the data to a VAX via ANSI labeled tape.

Name: Seadata

Version: 1.01 18-May-1990

Purpose: Seadata was written to replace the CARP and CRAP programs which were written to run on HP or LSI-11 computers.

Machine: 80286 or 80386 based IBM PC or compatible AT bus machine running MS-DOS version 3.2 or higher.

Language: Turbo C Version 2.0

Description: Seadata replaces the CARP and CRAP programs which were written to run on HP or LSI-11 computers respectively. The data is read from the Sea Data Model 12 cassette reader via a parallel interface (Intel 8255, 24-bit parallel I/O chip). The data is stored on input in one of three 2k byte buffers and reformatted into an output file which is a modification of the CARP format. The output file can be written to either virtual disk or hard disk. Once on disk, the files can be copied to network, tape, floppy, etc.

The function which this program provides was originally written for an HP computer and later rewritten for an LSI-11. The PC version provides some modifications:

1. Some of the Sea Data Model 12B readers have been modified to work only with an LSI-11. This modification was done to allow the LSI-11 to latch the data into its parallel port. The 8255 chip used in the parallel interface is fast enough to accept data from either version of the Sea Data reader.
2. The output buffers are much larger than before. The default output buffer size is 8k bytes. If smaller buffers are needed, the OUTB parameter in the include file seadata.h will need to be modified and the programs

recompiled and linked. Since the data is being written to disk, there may be some loss of performance and timing problems with smaller data buffers.

3. The LSI-11 version wrote characters indicating tape read errors to the terminal and/or printer. This version notes the type of errors on the monitor and prints a summary at the end of reading a tape.

4. There is no output to a printer while reading a cassette. A comment file, which summarizes the reading, is written to disk and can be printed after the reading is finished.

5. Some of the code needed for reading tapes with the model 0 or the 850 cartridge readers is in the seadata program; however, it has not been fully implemented or tested.

When seadata is executed, it first initializes its buffers and data areas and then prompts the user for the type of tape reader being used. The user must respond with a valid reader type before continuing. Next seadata paints four color bands on the screen. These bands are used to distinguish the usage of different parts of the screen. The top band is blue and is for permanent data about the program. The second band is black and is for prompting for information about the cassette being read. The third band is green and is used while reading the tape for a count of errors. The bottom band is brown and is used for error messages.

After this setup is completed, seadata prompts the user for the output file to use and opens the two files, comment and data. Next, seadata prompts for comments about the tape being read. Following the comments, seadata initializes the parallel input device and puts itself into a mode for collecting the data from the tape reader.

After the tape reading is finished, seadata writes the total number of tape records read and the number of errors both to the screen and to the comment file. Next it prompts the user for any further comments for the end of the tape reading.

At the end of reading a tape, seadata cycles to its beginning and asks the user if another tape is to be read. This cycle can continue until the tapes are finished or the disk is filled with data files.

Input: The seadata program is run by typing **"seadata"** at the DOS prompt. (assuming that seadata can be found in your path). While the program is running, it prompts the user for information. The requested prompts are discussed in the following paragraphs.

1. Seadata prompts for the reader type by printing:

   Please enter the type of reader you are using. Valid types are:

Model	*0 - 0*
Model	*12 - 12*
Cartridge	*850 - 850*

   Model #:

   The user should respond with the correct reader type. Valid entries are 0, 12, and 850. All other responses will be rejected.

   After the model number has been entered correctly, seadata will display this information as follows where **"xxx"** is the model number.

   Model # is xxx

2. Seadata prompts for the output file name by printing:

   Enter the root name of the file which you wish to use to store the data. The extensions '.DAT' and '.CMM' will be added for the data and comment files respectively.

   File name:

   At this point, the user needs to enter a name to use for storing the output data. The file can use any partition on the hard disk or on virtual disk. Space is a consideration as the output files may consume 2 to 3 megabytes for some of the longer tapes (between each tape, the program calculates the amount of free space on the partition just used). If a partition other than the default partition is to be used, it must be specified. The file name entered will have all characters to the right of any period (".") removed so the file name extensions can be added.

   If the file exists, seadata will ask if it is ok to write over the file. If not, then the user must specify a new file name or quit.

3. The third prompt requests the number of characters per tape record and is requested by printing:

   Enter the number of characters/cassette record (1-255):

   The user must enter a number greater than or equal to 1 and less than or equal to 255. If the entry is not in this range, the prompt will be reissued.

4. Comments are requested by printing:

   Enter comments - up to fifteen lines of information with a max. of 70 characters/line. End entry with an empty line (return only).

   1>

   The user can enter any comment information following the ">". The number of lines of comment is limited to 15 total. To end the comments, press the "enter" or "return" key at the ">" prompt.

5. To start the tape reading, seadata prompts the user as follows:

   Tape reading initialized.

   You may start the reader at any time.

   To stop the reading: Stop the reader, then

   press the ESC key.

   When this message is printed, the user may start the tape reader and the cassette records will be read. When the tape reading is finished, the user should stop the reader and then press the "ESC" or escape key. This will cause seadata to break out of its reading cycle, close the data file, and get additional comments for the end of the tape reading.

6. After reading a tape, seadata calculates the amount of free space on the disk partition just used and informs the user of the remaining space. Then it asks if another tape is to be read. The prompt is as follows where *"xxx"* is the byte count and *"Y"* is the partition designation:

   xxx bytes free on drive Y:

   Would you like to read another tape <yes or no>?

Output: The program produces two form of output, information displayed on the monitor and the data files on disk.

The information on the monitor is written during the tape reading and tells the user how many cassette records have been read and how many errors have been found during the tape reading. The display is printed after 1000 records have been read, after every 3000th record has been read, and after the last record has been read. The errors reported are of four different types:

1. Parity errors are reported when a problem is found with the longitudinal parity in the record. All of the data in the cassette record is written to the output buffer.

2. Long records have more characters in them than the user indicated. The data up to the user-specified record length is written to the output buffer and all extra data is discarded. A tape will usually have at least one long record at the beginning of the tape. A long record will frequently generate a parity error.

3. Short records have fewer characters in them than the user indicated. All of the data in the short record is kept and the record is padded to the end with binary 0's and written to the output buffer. Short records are excluded from parity checking.

4. Tape errors are unknown errors which were found by the tape reader and indicated to the seadata program. All of the data in these records are written to the output buffer.

The display of this information is in the green color band on the monitor and appears as follows with the x's indicating the numeric fields.

Cassette records: xxxxx

Processing errors:

Parity	*long records*	*short records*	*tape errors*
xxxxx	xxxxx	xxxxx	xxxxx

The disk files are a modification of the CARP format as documented in the program document by Mary Hunt in 1972. The modifications are of two types, a separate printable file for comments and larger data buffers.

The comment file (filename extension ".CMM") is a printable ascii file with the first record an ascii form of the CARP comment header. The format used is as follows where x's indicate numeric fields.

0FFA 4A4E xxx xxxxx xxxx xxx xxx xxx xxxxx

The fields, from left to right, are

Comment header indicator; = 0x0FFA,
Data record indicator; = 0x4A4E,
Number of 4 bit characters/cassette record,
Number of cassette records/block of output data,
Number of 16 bit words in each cassette record,
An archaic flag indicating 9 track tape = −1,
Flag indicating reader type; = 0 for model 0 reader,
= 1 for 850 cartridge reader, = −1 for Model 12 reader,
Flag indicating PC used for reading tape; = −1,
Length of the output buffers.

All records in the comment file following the header record are the text of the comments entered by the user and a summary of the number of records read and errors encountered.

The data file (filename extension ".DAT") contains the data and headers in the same format as documented in the original CARP documentation with the exception that the buffers are 8192 bytes in length. The format includes a header of 20 bytes at the beginning of each output buffer followed by the cassette records and their headers. The buffer header contains the following information:

bytes 0–1 Data record indicator; = 0x4A4E,
bytes 2–3 Expected number of 4 bit characters per cassette record,
bytes 4–5 The usual number of cassette records in an output buffer.

bytes 6–7 The number of 16 bit words per cassette record,

bytes 8–9 Sequential number of this output buffer,

bytes 10–11 Number of cassette records actually in this output buffer. This will usually equal the value in bytes 4–5,

bytes 12–13 Error indicator; nonzero if an error occurred while writing the previous output buffer to disk,

bytes 14–19 Not used.

Following the header, the cassette records are packed into the output buffer. Each cassette record has a 6 byte header followed by the data. The header for the cassette records has the following format:

bytes 0–1 Number of 4 bit characters actually found in this cassette record. This will usually equal the value in bytes 2–3 in the buffer header.

bytes 2–3 Error indicator word. The bits have the following meaning:

- bits 0–3 parity error if any bit = 1,
- bit 7 short record if = 1,
- bit 8 long record if = 1,
- bit 12 record error if = 1,

bytes 4–5 Sequential number of this cassette record. If greater than 65535, this number will wrap to zero.

Following the cassette record header, there are $n - 3$ words of data where n is the number in bytes 6–7 of the buffer header.

Space not used in an output buffer is filled with binary 0's.

Errors and Diagnostics: As with most programs, seadata can produce errors. Most of the messages which are displayed are in the user interaction portions of the program; however, some errors with significant impact on the tape reading can occur during the reading. The most significant messages and procedures for dealing with the problems are listed below.

A. Data entry errors:

1. *Input error - the model type must be specified as either 0, 12, or 850. Please reenter.*

2. *Input error - the model type must be 1-3 characters and specified as either 0, 12, or 850. Please reenter.*

   These two errors can occur if the user enters the wrong information for the type of tape reader being used. The user should re-enter the information. The only responses which are valid are "0", "12", and "850". Four incorrect entries will cause the program to terminate.

3. *Unable to read comment information - continue <yes or no>?*

   This message may appear if there is some problem with the program reading the information being entered for comments about the tape reading. Try the entry again; however, if the problem persists, break out of the program by typing ^C (control-C) and run it again.

4. *The number of characters in the cassette record must be* $>= 1$ *and* < 256.

   The number of characters in a cassette record must be greater than 0 and less than 256. If this message appears, there is a problem with the number entered. The number should be entered with no decimal point, just numeric characters.

B. File and I/O errors:

1. *Error opening output file*

2. *Open failure*

3. *Unable to open file*

   Would you like to try again <yes or no>

4. *Fatal file open failure - exiting!*

   These four error messages may occur as a part of the process of opening the output files for the comments and data. Some of the messages may also have DOS error messages. If one of these messages occurs, it would be best to check the amount of free space on the disk, check the name of the files you wish to use, or check for a hardware error. Then, run the program again.

5. **"file"** *already exists*

   Would you like to write over the file? <yes or no>

   If this message occurs, the program has found that the file already exists. The user is given the opportunity to write over the file with new data or enter a different file name.

6. *Error on write to disk file.*

7. *Unable to write comment information - continue <yes or no>?*

   These error messages refer to difficulties while writing information to the comment file. If possible, a DOS error message is also printed. There may be a hardware problem and it may be best to restart the program or the computer.

C. Data collection errors:

1. *Data overrun - input buffers overflowed.*

   Cassette record # nnn

   Buffer count = ccc

   Program will terminate.

This message occurs if the PC is not able to keep up with the tape reader and process the data as fast as it is sent to the PC. This might be caused by a number of factors; however, the most likely cause is a fragmented or full hard disk which requires extra movement of the disk heads. This causes the disk write to take longer than it should and the program cannot keep up with the reader. If this happens, seadata will terminate and will need to be restarted. The value "*nnn*" is the current cassette record number and the value "*ccc*" is the count of overflowed input buffers.

2. *Unable to write data to output file - status = xxx*

   This error will occur if there is a hardware problem with the writing of data to the hard disk. The program will attempt to continue; however, it may be best to restart the program. The value "*xxx*" is the status returned by the write routine.

3. *Error in record character count [>3 or <1].*

   This is one of those "should never happen" errors. If it does happen, there is probably an error within the tape reader since it is sending an incorrect count of characters in the cassette record.

4. *Unable to find end of long record (> 300 char.).*
   The tape reading will abort.

   This error will occur if the PC encounters a cassette record which is longer than expected. The program will scan up to 300 additional characters searching for the end of the record flag from the tape reader. If the end of record flag is not found, seadata assumes that the tape records are messed up beyond hope and will abort the tape reading. The limit for this loop can be changed in the module handle.c, function l_rec.

5. *Fatal tape read error - status = xxx*

   Exiting -

   This error message is written after all attempts at recovery have failed and seadata is about to terminate. The status value "xxx" can be traced back to the portion of the code where the failure occurred.

Programmer: Thomas W. Danforth

Date: 18-May-1990

Name: PCARP and PCARPHP

Version: 1.0 18-May-1990

Purpose: These two programs were written to convert files produced by reading Sea Data cassettes on a PC to a form usable by processing programs on a VAX.

Machine: VAX/VMS

Language: VAX C

Description: PCARP and PCARPHP were written to convert files produced by the seadata program on a PC to a form which can be used by processing programs on a VAX. PCARP will make the data appear to have been produced by the CRAP program (run on an LSI-11), while PCARPHP will make the data appear to have been produced by the original CARP program (run on an HP).

Both of these programs read the data files produced by the seadata program. The output is a file with either an ".LSI" or ".HP" extension (".LSI" for the CRAP look-alike and ".HP" for the CARP look-alike).

Each of the programs reads parameters entered on the command line. PCARP uses only one parameter which is the root for the file name while PCARPHP uses two, the first being the root for the file name and the second being the mode by which the data was transferred to the VAX.

After processing the command information, the programs open the input files (one data and one comment) and create the output file. Next they build the CARP or CRAP output files from the information in the comment file followed by the information in the data file. The CRAP format requires PCARP to copy the buffer header and the cassette record headers; however, the data in the cassette records must have the bytes swapped on output. The CARP format, on the other hand, requires PCARPHP to swap the bytes in the

buffer header and the cassette record headers while copying the data in the cassette records. PCARPHP also shortens the 8192 byte input buffer to 1600 bytes for output.

Input: Both of these programs require two forms of input. One is via the command line and the other is the data files from the seadata program.

PCARP and PCARPHP are located in the directory BUOY:[SOFT.RUN] and should be executed by defining them as foreign commands so that the parameters required for their execution can be passed on the command line. The foreign command definitions are:

PCARP == "$BUOY:[SOFT.RUN]PCARP.EXE"
PCARPHP == "$BUOY:[SOFT.RUN]PCARPHP.EXE"

When executing the programs, the user can simply enter the name of the program followed by any necessary parameters as described below.

To execute PCARP, the following command should be entered

PCARP file

where "**file**" is the parameter which specifies the root name of the files to be converted. The program searches this string for a period (".") and deletes anything beyond the first period found, assuming that this is the file name extension. Any directory or logical name must, therefore, not contain a period. The file extensions .CMM and .DAT will be added by the program for the input file names. The output file will have the same root name with the extension of .LSI. The input and output files will be in the same directory. If the output file already exists, a new version will be created.

To execute PCARPHP, the following command should be entered.

PCARPHP file access

where **"file"** and **"access"** are the two positional parameters passed to the program. **"file"** is the root name of the input data files as described for PCARP above. **"access"** is the route via which the data reached the VAX (access method). This will be either a 9-track tape written on the PC or ethernet. The user must enter either **"tape"** or **"enet"** on the command line. If not specified and the file name is specified, **"access"** will default to **"tape"**.

If no parameters are specified on the command line, the programs will go into interactive mode and prompt for the parameters. The prompt for the file name is

File name:

to which the user should provide the root for the input file names as described above. The prompt for the access method is

Access method:

to which the user should respond with either **"tape"** or **"enet"**. If the response to the prompt is incorrect, the program will terminate.

The input data files, one of comments from the tape reading and one of data, have the format described in the seadata program.

Output: PCARP and PCARPHP produce output files which can be used by processing programs on the VAX. PCARP's output file (extension ".LSI") is very similar to the file produced by the CRAP program on an LSI-11 with two differences. The first difference is the addition of the buffer length field in the comment header. This uses two bytes (# 16-17) of previously unused space. The second is the size of the data buffers. PCARP's output data buffers are 8192 bytes in length.

The output file for PCARPHP (extension ".HP") is almost identical to files produced by the CARP program on an HP computer. The only extension to the original definition is that the data records are fixed at 1600 bytes in length.

These programs produce no output to the terminal if they complete their operation correctly.

Errors and Diagnostics: PCARP and PCARPHP produce three classes of error messages. All of them are considered fatal.

1. *Error opening comment file fff*

   Error: nnn

   Program exiting

2. *Error opening data file fff*

   Error: nnn

   Program exiting

3. *Error opening output file fff*

   Error: nnn

   Program exiting

   These three error messages are written to SYS$OUTPUT if the program has a problem opening one of the files. The file name is substituted for "*fff*" and the error number is substituted for "*nnn*". The user should fix the problem and run the program again.

4. *Comment data scan error*

   Program exiting

   This error message is written if the program cannot decode the header line in the comment file (".CMM"). Since the remainder of

the processing is dependent upon this information, the program terminates. The user should check to be sure that the file was transferred to the VAX correctly and that, if using PCARPHP, that the correct access method is specified.

5. *Error during read - nnn bytes read*

6. *Error during write - nnn bytes written*

   These two error messages are written if there is a problem either reading or writing one of the data files. The number of bytes read or written is reported as "*nnn*" as a possible indication of the problem or where the problem occurred. Both of these messages will be followed by the system error message which may be of assistance in fixing the problem.

Programmer: Thomas W. Danforth

Date: 18-May-1990

Appendix 1: 24 bit parallel interface from Optimal Technology, Inc.

The parallel interface used by seadata for reading tapes was purchased from Optimal Technology, Inc., Rt. 1 - Box 138, Earlysville, Virginia 22936. The price was $89.00 in 1989. The interface uses an Intel 8255 programmable, parallel processor chip.

In order for the Sea Data Model 12 reader to work with the PC, the strobe signal from the Model 12 had to be inverted so it would be recognized by the 8255 processor. For the data reading, pin #1 on J1 on the IB-24 is used to get the strobe signal from the reader (see schematic in Figure 1). The trace which connects this pin to bit PC7 on the 8255 was cut and the signal passed to a transistor (2N3904) to invert it. The output from the transistor was then wired to pins #3 & #14 on J1 on the IB-24. These are connected to bits PC2 and PC4 which are the strobe lines for units B & A, respectively, in the 8255.

The IB-24 was also modified to pass an interrupt signal from the 8255 processor to the AT bus. The interrupt used was IRQ 3, the COM 2 interrupt. The interrupt signal was wired to the AT bus by connecting a wire from pin #10 on J2 to bus line B25.

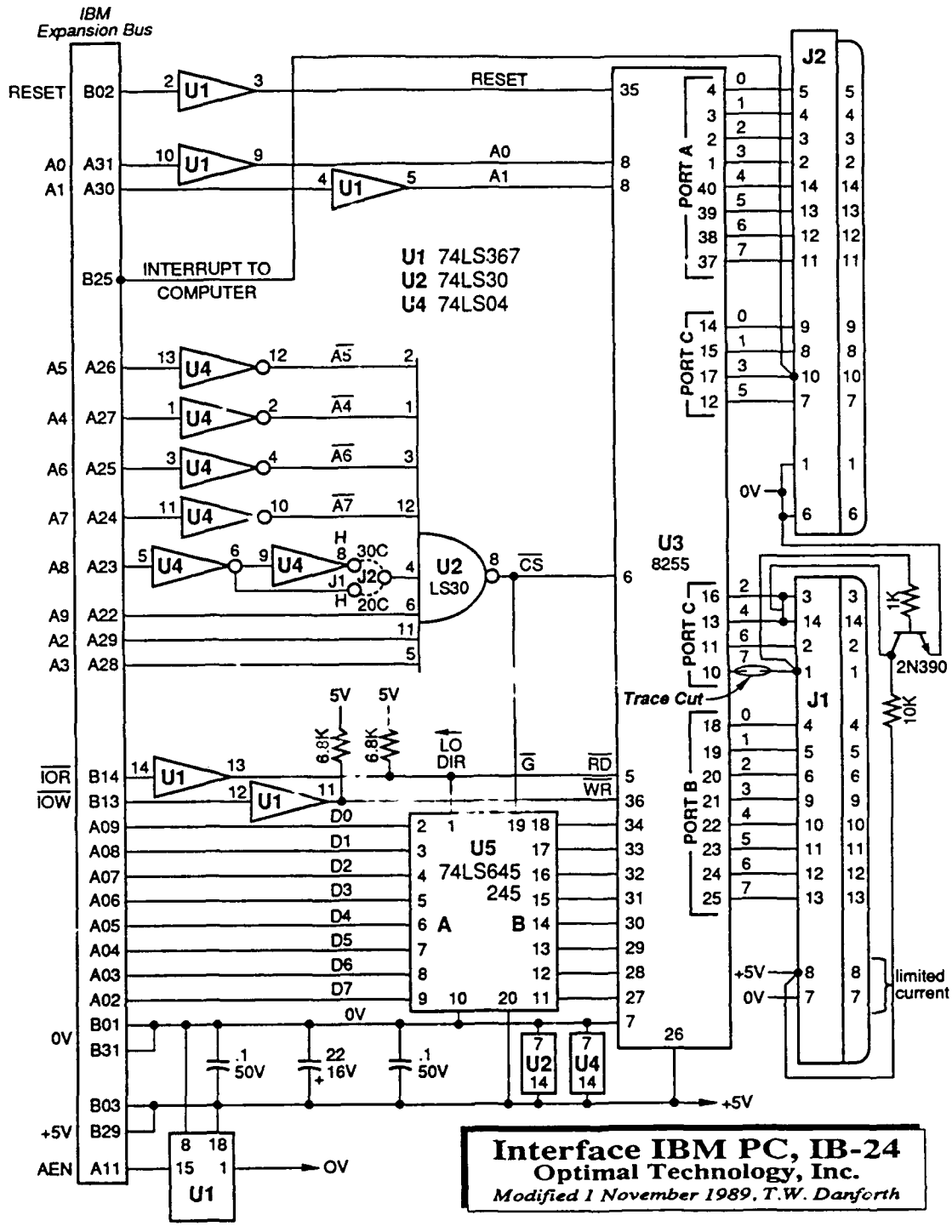

IBM
Expansion Bus
RESET
B02
A0
A31
A1
A30
B25
INTERRUPT TO
COMPUTER
U1 74LS367
U2 74LS30
U4 74LS04
A5
A26
A4
A27
A6
A25
A7
A24
A8
A23
A9
A22
A2
A29
A3
A28
U2
LS30
CS
U3
8255
PORT A
PORT B
PORT C
J2
J1
Trace Cut
2N390
1K
10K
IOR
IOW
B14
B13
LO
DIR
G
RD
WR
D0
D1
D2
D3
D4
D5
D6
D7
U5
74LS645
245
A
B
B01
B31
B03
B29
A11
AEN
0V
+5V
.1
50V
22
16V
limited
current
Interface IBM PC, IB-24
Optimal Technology, Inc.
Modified 1 November 1989, T.W. Danforth

Figure 1

Appendix 2: Cabling documentation

A cable was designed for the PC to Model 12 reader which would put the data signals in the best order for the PC to use them. This is not the same as documented in the Model 12 reader documentation. The order of the data bits in the PC is as follows.

bit 15 – bit 12	Highest data "character"
bit 11 – bit 8	Middle data "character"
bit 7 – bit 4	Lowest data "character"
bit 3	end of record bit
bit 2	character count – high bit
bit 1	character count – low bit
bit 0	error flag

The IB-24 interface has two 14 pin connectors. Each is configured to use one of the data ports on the 8255 parallel chip (either A or B) and half of port C. Seadata uses ports A and B for data and uses the signals in port C for strobing the data into the 8255 chip and then interrupting the computer.

The wiring necessary to connect the IB-24 interface with J1 on the Model 12 reader is shown in Table 1.

Table 1: Wiring to connect the IB-24 interface with J1 on the Model 12 reader.

IB-24 function	Pin	25 pin connector	wire color	Model 12 - J1 Pin #	Model 12 - J1 function
J2					
A3	2	13	blue - blk	11	DL 7
A2	3	12	orng - blk	10	DL 6
A1	4	11	blk - wht	9	DL 5
A0	5	10	orng	8	DL 4
C5					
C1					
C0					
A7	11	22	wht	15	DL 11
A6	12	23	green	14	DL 10
A5	13	24	red	13	DL 9
A4	14	25	red - grn	12	DL 8
	6	9	wht - blk,red	33	ground
J1					
C7	1	4	orng - grn	17	strobe
C6					
B0	4	3	blk	1	message
B1	5	2	blk - wht,red	2	WL0
B2	6	1	wht - blk	3	WL1
B3	9	14	red - wht	16	Last
B4	10	15	grn - wht,blk	4	DL0
B5	11	16	blue	5	DL1
B6	12	17	red - wht,blk	6	DL2
B7	13	18	blue - red	7	DL3

Appendix 3: Ethernet file copies

One of the most efficient methods of copying the seadata output files from a PC to a VAX is via ethernet using ftp (file transfer program). When using ftp, you must remember that the .CMM files are printable ascii and the .DAT files are binary. Therefore, you must use different methods of transfer within the ftp program, ascii for the .CMM files and binary for the .DAT files.

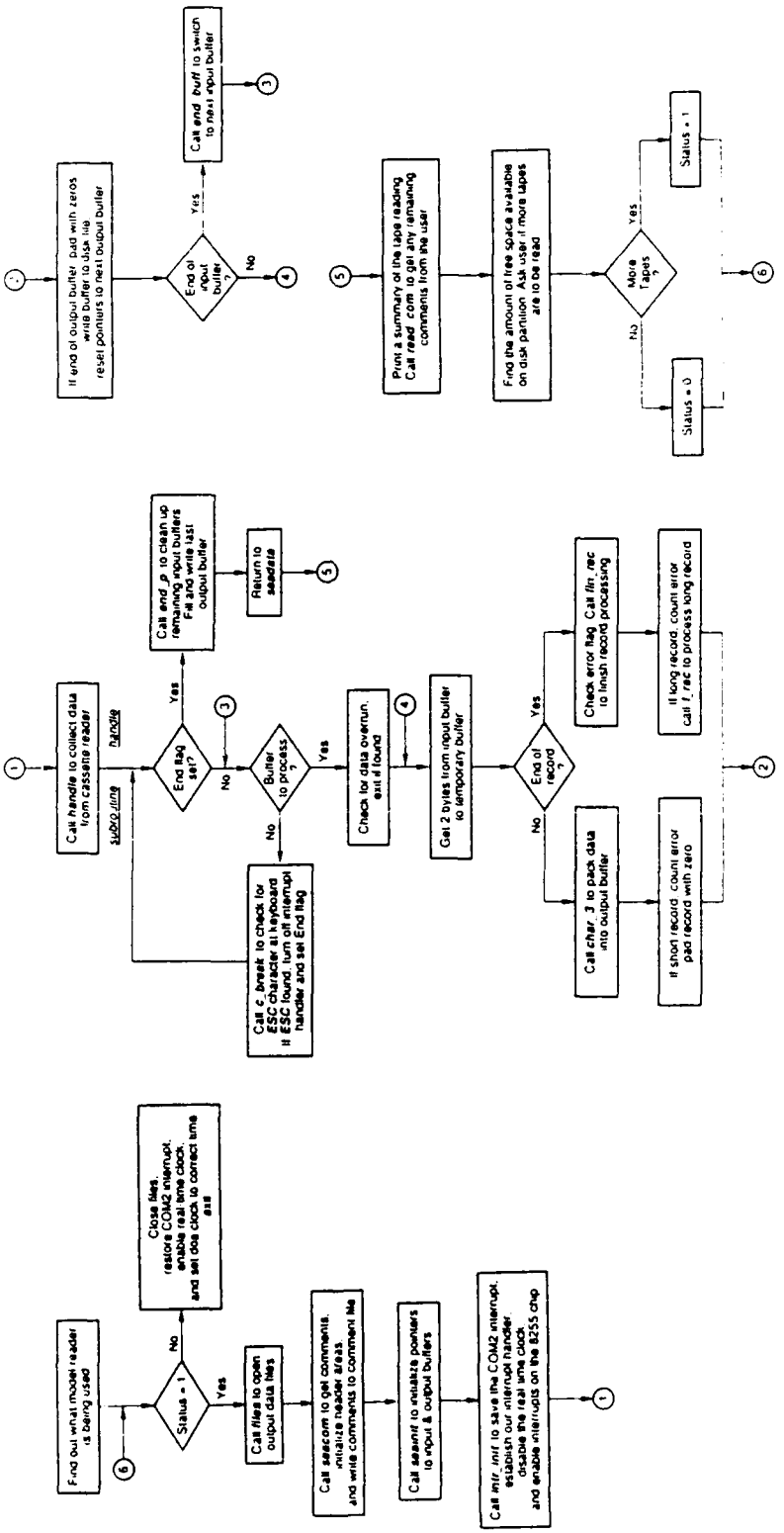
The file transfers can be accomplished as shown below. For a complete discussion of ftp, you should read a TCP/IP or ftp manual. To start ftp, you should simply enter ftp (assuming that ftp is available on your PC and can be found in your PATH).

ftp system	# system is the destination of the files.
Remote User Name:	# you enter your user name and password
Remote Password:	# for the destination system.
ftp> **cd direct**	# ftp responds with its prompt. The user can
	# enter commands to change directory and copy
	# files. A selection of useful commands is
	# given below.
ftp> **binary**	# Change to binary file copy.
ftp> **put file.DAT**	# Copy (put) the data file to the destination.
ftp> **ascii**	# Change to ascii file copy.
ftp> **put file.CMM**	# Copy the comment file to the destination.
ftp> **bye**	# Logoff destination system and exit ftp.

Useful commands include the following.

cd remote_directory	Change working directory to remote_directory.
pwd	Print name of remote working directory.
prompt	Toggle between prompting and no prompting for copying multiple files.
mput local_file	Copy one or more local_files to the remote working directory.
binary	Change file transfer mode to binary.
put local_file	Copy local file to the remote working directory.
ascii	Change file transfer mode to ascii.
bye	Logoff remote system and exit ftp.

Appendix 4: Flow diagram for seadata program



Appendix 5: Program listings

Programs are listed alphabetically.

```
#include <conio.h>
#include <stdio.h>
#include <dos.h>
#include "seadata.h"

/*      cbreak.c
/*      T.W. Danforth           15-Feb-1990
/*         TWD  18-May-1990
/*              Reset the three data ports (A, B, C) in the 8255 following
/*              the collection of data.  The reset puts the three ports
/*              to mode 0 with all set for input (high impedance on the
/*              lines).
/*
/*      This file contains two routines used for handling interrupts or
/*      breaks in the normal stream of data in the Seadata collection.
/*      The first routine checks for an ESCape character being entered on
/*      the keyboard.  The second is Sea_int which is the interrupt routine
/*      to get data from the 8255 parallel input processor.
  */

/*      c_break
/*      Routine to capture a ESC, write an ending message, and turn off
/*      the collection of data from the cassette reader.  The end of
/*      data collection is marked by setting  end_proc = 1.
/*
/*      Parameters:
/*              none.
/*
/*      Return values:
/*              none.
  */

void    c_break (void)
{
        extern  BUFF    D_buff;         /* data buffers */
        void    getrtime (int *), setdtime (int*);

        unsigned char i_mask;
        int     dtm[6];

        if (kbhit ()  &&  getch() == 0x1b)      /* if ESC char., then end */

           {  fflush (stdin);           /* attempt to clean out chars. */

              setvect (COM2, D_buff.oldfunc);   /* Restore COM2 interrupt. */
              outportb (CONTROL, 0x9B);         /* Reset the 8255 processor */

/*              Enable the real-time clock interrupt and disable the
/*              COM2 interrupt.
  */
              i_mask = inportb (0x21) | 0x08;   /* Get mask and reset COM2 */
              outportb (0x21, (i_mask & 0xFE)); /* Set mask and enable IRQ 0 */
```



```
/*              Get the value in the real-time clock and reset
/*              the DOS clock.
  */
              getrtime (dtm);           /* Get real_time clock value */
              setdtime (dtm);           /* set DOS clock */

              D_buff.L_count = D_buff.Rawpoint - D_buff.r_point[D_buff.which_raw];
              D_buff.inb_count ++;
              D_buff.end_proc = 1;

           }
        return ;
}


/*      Sea_int
/*      This routine removes the bytes of data from the 8255 parallel
/*      input ports.  The high order byte is comming in Port A and the
/*      low order byte is comming in Port B.
  */

void    interrupt Sea_int ()
{
        extern  BUFF    D_buff;         /* the data pointers and storage */

        *(D_buff.Rawpoint) = inportb (PORT_A);          /* get Port A */
        D_buff.Rawpoint ++;

        *(D_buff.Rawpoint) = inportb (PORT_B);          /* get Port B */
        D_buff.Rawpoint ++;

/*              Check for the end of the input buffer.  If found
/*              we need to switch to the next buffer.
  */
        if (D_buff.Rawpoint >= D_buff.Raw_end)
           {  D_buff.inb_count ++;              /* another buffer is ready */
              if (D_buff.which_raw == 2)
                 {  D_buff.which_raw = 0;  }
              else
                 {  D_buff.which_raw ++;  }

              D_buff.Rawpoint = D_buff.r_point[D_buff.which_raw];
              D_buff.Raw_end = D_buff.Rawpoint + D_buff.R_count;
           }

        outportb (CONTROL, 0x09);       /* Set Interrupt enable again, just in case */
        outportb (0x20, 0x20);          /* Set the End-of-Interrupt bit  */
                                        /* so normal processing can continue */

        return;
}
```

```
#include <dos.h>
/*      diskfr.c
/*      T.W. Danforth           14-Mar-90
/*      Routine to find the number of free bytes on the storage
/*      device being used for the Seadata cassettes.
/*
/*      Parameters:
/*              *disk   - Pointer to a character which contains the
/*                        upper case character for the designating the drive.
/*
/*      Returns:
/*              bytes   - a long integer which is the number of free bytes.
  */

long    disk_free (char *disk)
{
        typedef struct  {
                unsigned int    avail;
                unsigned int    total;
                unsigned int    bsec;
                unsigned int    sclus;
                }  dfree;
        dfree   df;

        unsigned char   d;
        unsigned long   bytes;

        d = *disk;
        d -= 64;                /* first char starts at 1 */
        getdfree (d, (struct dfree *) &df);

        bytes = (long) df.bsec * (long) df.sclus * (long) df.avail;
        return (bytes);
}
```

```
#include <conio.h>
#include <stdio.h>
#include "seadata.h"

/*      endp.c
/*      T.W. Danforth           15-Feb-1990
/*      Routine to finish the data processing of the Seadata cassette.
/*      This is run after the collection program has been signaled to
/*      stop by the user typing an <ESC>.
/*
/*      Parameters:
/*              fh_d    - file handle for the output data file.
/*              count   - The count of characters processed in the
/*                        current data record.
/*
/*      Return values:
/*              none.
  */

void    end_p (int fh_d, int count)

{

        extern  BUFF    D_buff;         /* Data buffers and pointers */
        extern  D_Head  Datahead;       /* Header area for output. */
        extern  C_Head  Comhead;        /* Comment headers. */
        extern  C_field Comment;        /* Comment array. */
        extern  R_Head  Rechead;        /* Headers. */

        int     i, j, temp, istat;
        int     c_count;                /* Count of characters in the record */

        int     l_recl ();
        void    end_buff (), e_line ();
        void    fin_rec (int, short int, int);  /* Finish record processing. */
        void    r_pnt ();

        struct  sl      {
                unsigned char lo;
                unsigned char hi;
                };
        union   {
                short   int     word;   /* Input word from cassette reader */
                struct  sl      ch;     /* Input word as bytes. */
                } inp;
```



```
        static  char    Modul[]={"end_p  --  "};
        static  char    Err4[]={"Program will terminate."};
        static  char    Err6[]=
                  {"Unable to write data to output file - status = "};

/*              Initialize counter.     */
        c_count = count;                /* Count characters in input */

/*              Process the full input buffers.  Loop until the
/*              count of the input buffers = 0.
  */
        while (D_buff.inb_count > 0)
        {
        if (D_buff.inb_count == 1)
            D_buff.R_count = D_buff.L_count;   /* Size - last buffer. */

        D_buff.inb_count --;
        D_buff.i_count = 0;     /* Count bytes removed from input */

/*              Loop to process the bytes until the end of the buffer.
  */
        while (D_buff.i_count < D_buff.R_count)
           {  inp.ch.hi = *(D_buff.Procpoint)++;
              inp.ch.lo = *(D_buff.Procpoint)++;  /* Get input word */
              D_buff.i_count += 2;

/*              Automatic add of 3 chars. since this is probably not the
/*              end of a record.  If this is incorrect, then fix it later.
  */
              c_count += 3;
              temp = 3;

/*              Check the character count.  If it indicates that we
/*              have the correct number or more than the correct number
/*              of characters, then begin the end_of_record processing.
/*              If we still have less than the full record, continue
/*              under else.
  */
              if (c_count >= D_buff.r_c_count)

/*              Normal end of record processing.
/*              Figure how many characters in the record and add that
/*              to the count.  Then check for any errors and go
/*              finish the record processing.
/*              If D_buff.adex ==1, this is model 12, use 3 characters.
  */
               {  if (D_buff.adex != 1  &&  (inp.ch.lo & END) == END)
                     {  temp = (int) inp.ch.lo & CHAR;  /* char. count */
                        temp = temp >> 1;       /* Shift into place. */
                        c_count -= 3;           /* remove constant */
                        c_count += temp;        /* add characters. */
                     }
```

```
            if ((inp.ch.lo & ERRF) == ERRF)   /* check errors */
              {  Rechead.R_head.errind = Rechead.R_head.errind | ERROR;
                 D_buff.e_error ++;
              }
/*          Finish record processing. */
            fin_rec (temp, inp.word, c_count);

/*          Check to see if this is a long record and set
/*          error flag if necessary.
  */
            if ((inp.ch.lo & END) != END)
               {  if (l_recl () == 1)
                     return;    /* Quit if error in long rec. */
               }
            c_count = 0;
            r_pnt ();           /* copy header - reset pointers */
         }
      else

/*          This is probably not the end of the input record.
/*          If end is indicated (END bit set), finish record and fill,
/*          else, move only 3 characters to output.
  */
          {  if ((inp.ch.lo & END) != END)
                char_3 (inp.word, 0);    /* move full 3 char. */


/*          Short record.          */
/*          If D_buff.adex ==1, this is model 12, use 3 characters.  */
            else
               {  if (D_buff.adex != 1)
                   { temp = (int) inp.ch.lo & CHAR;  /* char. count */
                     temp = temp >> 1;  /* Shift into place. */
                     c_count -= 3;      /* remove constant */
                     c_count += temp;   /* add characters. */
                   }

/*          Check to see if the strange 3 character message word is
/*          being sent as a record.  If true, the char. count will be
/*          3 and the high order byte will be 0xf0.  Skip this junk
/*          and go process the next record.
  */
                  if (c_count == 3)
                     {  c_count = 0;
                        D_buff.m_error ++;  /* Flag message word */
                        D_buff.p_value = 0; /* clear counters. */
                        D_buff.parity.storage = 0;
                        D_buff.move_flag = 0;
                     }
                  else
```



```
                  {  Rechead.R_head.errind =
                        Rechead.R_head.errind | SHORTR;
                     D_buff.s_error ++;      /* short error */
                     fin_rec (temp, inp.word, c_count);
                     for (; c_count < D_buff.r_c_count;
                                 c_count += 2)
                        {  *(D_buff.Outpoint) = 0x00;
                           D_buff.Outpoint++;
                        }
                     c_count = 0;
                     r_pnt ();       /* reset pointers. */
                  }
            }
      }

/*          If the output buffer is full, then empty it and
/*          reset pointers to the next buffer.  First, pad the end with
/*          0x00 and fill in the header information.
  */
   if (Datahead.header.ncas >= Datahead.header.rec_tape)
      { for (; D_buff.Outpoint < D_buff.Out_end; D_buff.Outpoint++)
           *(D_buff.Outpoint) = 0x00;
        memmove (D_buff.o_point[D_buff.which_out],
                    Datahead.string, 20);
        istat = write (fh_d , D_buff.o_point[D_buff.which_out],
                       OUTB);
        if (istat != OUTB)
           {  e_line ();
              cprintf ("%s%s%d", Modul, Err6, istat);
              e_line ();
              cputs (Err4);
             Datahead.header.errind = istat;
           }
        else
           Datahead.header.errind = 0;

        if (D_buff.which_out == 0)
           {  D_buff.which_out = 1;  }
        else
           {  D_buff.which_out = 0;  }

        D_buff.Outpoint = D_buff.o_point[D_buff.which_out] +
             13 * sizeof (short);
        D_buff.rec_p = D_buff.o_point[D_buff.which_out] +
             10 * sizeof (short);
        D_buff.Out_end =D_buff.o_point[D_buff.which_out] +
             OUTB * sizeof (char);

        Datahead.header.nrec ++;
        Datahead.header.ncas = 0;
      }
```

```
/*              Check for the end of the input buffer.  If found then,
/*              reset pointers and go back for more data.
  */
              if (D_buff.Procpoint >= D_buff.Proc_end)
                 {  end_buff ();
                    D_buff.i_count = D_buff.R_count; }
           }
        }
        return;
}


/*      i_recl
/*      Routine to handle a long record.  This includes the copying
/*      of the data to the work area, counting the bytes, checking for the
/*      end of the input buffer.  This routine is used only for the end
/*      of the data input processing.
/*
/*      Parameters:
/*              none
/*
/*      Return values:
/*              0       - routine processed ok.
/*              1       - error in processing.
  */

int     i_recl ()
{

        extern  BUFF    D_buff;
        extern  D_Head  Datahead;
        extern  R_Head  Rechead;

        int     i;

        union   {
                struct  {                       /* bytes in data word */
                        unsigned char s1;       /* high byte */
                        unsigned char s2;       /* low byte */
                        } b;
                short   int  t;                 /* data word */
                } tmp;

/*              Flag the long record.           */
        Rechead.R_head.errind = Rechead.R_head.errind | LONGR;
        D_buff.i_error ++;
```



```
/*              Read in bytes of data and check for the end of record.
/*              Loop through this for a max. of 200 bytes.
  */
        for (i = 0; i < 200; i += 2)
           {  tmp.b.s2 = *(D_buff.Procpoint)++;
              tmp.b.s1 = *(D_buff.Procpoint)++;
              D_buff.i_count += 2;      /* count bytes removed */
              Rechead.R_head.nchar += 3;  /* count characters */

              if ((tmp.b.s1 & END) == END)  /* If end found, */
                 {  return (0); }       /* set status - return. */

/*              If we find the end of the input buffer, then we need
/*              to end the processing and return to the user.
  */
              if (D_buff.Procpoint >= D_buff.Proc_end)
                 {  return (1);  }
           }

/*              If we end this loop, then we have checked 200 bytes
/*              (approx. 300 char.) and not found the end of this
/*              record.  This is a hopeless situation and we need
/*              to abort the data collection.
  */
        return (1);
}
```

```
#include <dir.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <errno.h>
#include <io.h>
#include <conio.h>
#include "seadata.h"
#include <ctype.h>

/*      files.c
/*      T.W. Danforth           26-aug-89
/*      Function to obtain file name and open file requested by user.
/*      This function does some errror control and access checking.
/*      If a file can not be opened, the function will force the exit
/*      of the program.
/*
/*      Parameters:
/*              *fh1    - Address of where the file handle for the data
/*                        file is to be stored.
/*              *fh2    - Address of where the file handle for the comment
/*                        file is to be stored.
/*              *dr     - Pointer to character which will contain the
/*                        drive designator.
/*
/*      Returns:
/*              The file handle values are stored in fh1 and fh2.
/*              status = 1      ok.
/*                       -3     Failure opening one of the files and the
/*                              user does not want to try again.
  */

int     files (int *fh1, int *fh2, char *dr)
{
        extern  BUFF    D_buff;
        void    e_line (void), p_line (void), t_line (void);

        char    driv, *cn;
        char    *path, *p, file[70];
        int     fh, jj, kk;
        register int ans;
        int     filopen (char *, int), fill (char *);
        static  char *ext[2]={".DAT",".CMM"};

        path = &file[2];
        file[0] = 68;
```



```
        D_buff.prm_line = 0;
        p_line ();
        clrscr ();
        cputs
         ("Enter the root name of the file which you wish to use to store");
        p_line ();
        cputs
         ("the data.  The extensions '.DAT' and '.CMM' will be added for the");
        p_line ();
        cputs ("data and comment files respectively.");

/*              Loop until the files are open or we return to the calling
/*              routine.
  */
        kk = 0;
        while (kk == 0)
/*              Get the file name base for opening the files.
  */
        {  p_line ();
           cputs ("File name: ");
           cgets (file);

/*              Scan for a '.' and, if not found, then add '.DAT' and
/*              '.CMM' for the files.
  */
           p = strchr (path, '.');
           if (p == 0)                  /* Point to file extension */
              p = &file [file [1] + 2];
           jj = 0;

/*              Add the '.DAT' or the '.CMM' and then call
/*              the open routine.
  */
           D_buff.top_line = 1;
           while (jj < 2)
           {  strcpy (p, ext[jj]);
              t_line ();                /* Note name of file. */
              cprintf ("Opening file: %s", path);
              fh = filopen (path, jj);

              if (fh <= 0)              /* if < 0, something wrong. */
                {  if (fh == -4)
                      break;            /* don't write over file */
                   if (fill (path) == 1)
                      break;            /* try again. */
                   else
                      return (-3);      /* failure - quit */
                }
```

```
            if (jj == 0)
              {  *fn1 = fn;
                 jj = 1; }
            else
              {  *fn2 = fn;
                 jj = 3;
                 kk = 3; }
         }
      }
/*          Before returning, find the character designating the
/*          device being used for holding the data.
  */
      ch = strchr (path, ':');
      if (ch == NULL)
         {  driv = (unsigned char) getdisk ();  /* default disk? */
            *dr = driv + 65;          /* make into a char */
         }
      else
         {  ch--;             /* point to the char. before ':' */
            *dr = (char) toupper (*ch); }
      return (1);
}
```



```
/*      fill
/*      Routine to print error message and check on whether the
/*      user wishes to continue to try to open a file.
/*
/*      Parameters:
/*              p       - Pointer to the path of the file being opened.
/*      Returns:
/*              -3      - Failure to read correctly from the keyboard.
/*              1 or 2  - The status from yno.
  */
int     fill (char *p)
{
        extern  BUFF    D_buff;
        void    e_line (void);
        int     j, ans;
        int     yno (void);

        j = 0;
        while (j < 4)
        {  D_buff.er_line = 0;
           e_line ();
           clrscr ();
           cprintf ("Unable to open %s", p);
           e_line ();
           cputs ("Would you like to try again <yes or no> ");

           ans = yno ();        /* get and check the answer */
           if (ans != 0)
               return (ans);
           j++;
        }
        return (-3);            /* failure to read correctly */
}
```

```
/*      filopen
/*      Routine to open the output data and comment files.  This uses
/*      a pointer to the file name as input and returns the integer
/*      file handle as the output.
/*
/*      Parameters:
/*              *path   - Pointer to the file's path name string.
/*              flg     - Value indicating type of file to open.
/*
/*      Returns:
/*              >0      - The file handle of the opened file.
/*              -3      - An open failure.
/*              -4      - User chose not to write over this file.
  */

int     filopen (char *path, int flg)

{
        extern  BUFF    D_buff;
        extern  int     errno;
        extern  char    *sys_errlist[];

        void    e_line (void), p_line(void);

        int     open(const char *, int,...), yno(void);
        int     ans;
        int     fh, flag2;
        static  int flag = (O_RDWR|O_TRUNC);
        static  int flags = (O_RDWR|O_CREAT|O_EXCL);

        static  char Err1[]={"Error opening output file"};
        static  char Err3[]={"Open failure"};

/*              Set up the open flags based on the flag.
  */
        if (flg == 0)
            flag2 = O_BINARY;
        else
            flag2 = O_TEXT;

/*              Try to open file.  If it exists, return error -
/*              deal with later.  Open is for read/write access,
/*              binary file.
  */
        fh = open (path, flags | flag2, S_IREAD|S_IWRITE);
```



```
/*              Check open errors.  If fh > 0 then open worked ok.
/*              If fh <= 0, then errors which we need to deal with.
  */
        if (fh <= 0)
           { switch (errno) {
             case EEXIST:       /* File exists. Open anyway? */
                  ans = fol (path);     /* print error */

/*              Check the answer - if = 1, then open with truncate
/*              if = 0 or 2, then we need to open a different file.
  */
                  if (ans == 1)
                    {  fh = open (path, flag | flag2);
                       if (fh <= 0)
                         {                  /* Print error message */
                           e_line ();
                           cputs (Err1);
                           return (-3); }
                       else
                            return (fh);
                    }
                  else
                    {  e_line ();       /* don't open this file */
                       cputs (Err3);
                       return (-4); }

                        /* No file or directory or Perm. denied
                        /* User will need another file name
                          */
                 case ENOENT:
                 case EACCES:
                    e_line ();
                    clrscr ();
                    cputs (sys_errlist[errno]);
                    e_line ();
                    cputs (Err3);
                    return (-3);

                        /* Case of all other possible errors.
                        /* Just issue an error message and exit.
                          */
                 default:
                    e_line ();
                    cputs (sys_errlist[errno]);
                    return (-3);
                 }
              }

        return (fh);            /* The file is open correctly. */
}
```

```
/*      fol
/*      Routine to check response of the user on open failure.
/*      User must respond with "yes" or "no" before program will
/*      continue.
/*
/*      Parameters:
/*              path    - pointer to the file's path.
/*      Returns:
/*              response = 1 if yes
/*                       = 2 if no
  */
int     fol (char *path)
{
        extern  BUFF    D_buff;
        void    e_line (void);
        int     i, ans, yno (void);

        i = 0;
        while (i == 0)
        {  D_buff.er_line = 0;
           e_line ();
           clrscr ();
           cprintf ("%s already exists", path);
           e_line ();
           cputs
            ("Would you like to write over the file? <yes or no> ");
           ans = yno();

/*              Check answer - if = 1 or 2, open return to calling
/*              routine with the response.  If = 0, restate the
/*              error and ask question again.
  */
           if (ans == 1 || ans == 2)
              return (ans);
        }
}
```

```
#include <mem.h>
#include <conio.h>
#include <stdio.h>
#include "seadata.h"

/*      handle.c
/*      T.W. Danforth           15-Feb-1990
/*      Routine to handle the data which comes in the parallel interface
/*      from the SeaData reader.  This routine copies all of the data
/*      from the raw input buffers, checks for short or long records,
/*      checks for parity errors, and removes the flag bits.  The data
/*      is then packed into an output buffer for writing to the output
/*      disk or memory file.
/*
/*      Parameters:
/*              fh_d    - file handle for the output data file.
/*
/*      Return values:
/*              = 0 - successful tape reading.
/*              != 0 - some subroutine failed.
  */

int     handle  (int fh_d)

{
        extern  BUFF    D_buff;         /* Storage for data buffers */
        extern  D_Head  Datahead;       /* Header area for output . */
        extern  C_Head  Comhead;        /* Comment headers. */
        extern  C_field Comment;        /* Comment array. */
        extern  R_Head  Rechead;        /* Headers. */

        int     i, j, temp, istat, jstat;
        int     c_count;

        int     i_rec ();
        void    e_line (), c_break (void);
        void    end_buff (void), end_p (int, int);
        void    fin_rec (int, short int, int);
        void    r_pnt ();

        struct  sl      {
                unsigned char lo;
                unsigned char hi;
                };
        union   {
                short   int     word;   /* Input word from reader */
                struct  sl      ch;     /* Input word as bytes. */
                } inp;
```



```
        static  char    Modul[]={"handle  --  "};
        static  char    Err1[]=
                  {"Data overrun  --  input buffers overflowed."};
        static  char    Err2[]={"Cassette record # "};
        static  char    Err3[]={"Buffer count = "};
        static  char    Err4[]={"Program will terminate."};
        static  char    Err6[]=
                  {"Unable to write data to output file - status = "};

/*              Initialize counters.    */

        c_count = 0;    /* Count characters in input records */
        D_buff.inb_count = 0;
        D_buff.move_flag = 0;
        D_buff.p_value = 0;

/*              Unpack data until end processing flag set
  */
        D_buff.end_proc = 0;
        while (D_buff.end_proc != 1)
        {

/*              Check for a full input buffer.  If the count = 0,
/*              then there are none full  --  wait.
/*              If count != 0, then start to process.
/*              Check for data overrun first.
  */

        while (D_buff.inb_count <= 0)
           {  c_break (); }     /* Check for end of data */

        if (D_buff.end_proc)    /* Provide break incase this is */
           continue;            /* last buffer to be processed. */

        if (D_buff.inb_count > 3 )  /* Data overrun - ERROR exit */
           {  D_buff.er_line = 0;
              e_line ();
              cprintf ("%s%s", Modul, Err1);
              e_line ();
              cprintf ("%s%lu", Err2, D_buff.records);
              e_line ();
              cprintf ("   %s%d", Err3, D_buff.inb_count);
              e_line ();
              cputs (Err4);
              return (10);
           }

        D_buff.inb_count --;
        D_buff.i_count = 0;     /* Count bytes removed from input */
```

```
/*              Loop to process the bytes until the end of the buffer.
  */
        while (D_buff.i_count < D_buff.R_count)
           {  inp.ch.hi = *(D_buff.Procpoint)++;
              inp.ch.lo = *(D_buff.Procpoint)++;  /* Get input word */
              D_buff.i_count += 2;

/*              Automatic add of 3 chars. since this is probably not the
/*              end of record.  If incorrect, then fix later. */
              c_count += 3;
              temp = 3;

/*              Check character count.  If it indicates the
/*              correct number or more characters, then begin
/*              end_of_record processing.  If less than full record,
/*              continue under else.  */
              if (c_count >= D_buff.r_c_count)

/*              Normal end of record processing.  Figure characters
/*              in record and add to the count.  Check for errors and
/*              go finish record processing.
/*              If D_buff.adex ==1, this is model 12, use 3 characters.
  */
               {  if (D_buff.adex != 1  &&  (inp.ch.lo & END) == END)
                     {  temp = (int) inp.ch.lo & CHAR;  /* char. count */
                        temp = temp >> 1;       /* Shift */
                        c_count -= 3;           /* remove constant */
                        c_count += temp;        /* add characters. */
                     }
                  if ((inp.ch.lo & ERRF) == ERRF)   /* check errors */
                     {  Rechead.R_head.errind =
                           Rechead.R_head.errind | ERROR;
                        D_buff.e_error ++;
                     }
                  fin_rec (temp, inp.word, c_count);

/*              Check to see if this is a long record and set
/*              error flag if necessary.  */
                  if ((inp.ch.lo & END) != END)
                     {  jstat = i_rec ();
                        if (jstat != 1)       /* return if */
                            return (jstat);   /* failure. */
                     }
                  c_count = 0;
                  r_pnt ();           /* reset pointers */
               }
              else
```



```
/*              This is probably not the end of the input record.
/*              If end is indicated (END bit set), finish record and fill,
/*              else, move only 3 characters to output.
  */
               {  if ((inp.ch.lo & END) != END)
                     char_3 (inp.word, 0);    /* move full 3 char. */

/*              Short record.           */
/*              If D_buff.adex ==1, model 12 , use 3 characters.  */
                  else
                     {  if (D_buff.adex != 1)  /* char. count */
                        {  temp = (int) inp.ch.lo & CHAR;
                           temp = temp >> 1;      /* Shift */
                           c_count -= 3;          /* remove constant */
                           c_count += temp;       /* add characters. */
                           }

/*              Check to see if the strange 3 character message word is
/*              being sent as a record.  If true, the char. count will be
/*              3 and the high order byte will be 0xf0.  Skip this junk
/*              and go process the next record.  */
                        if (c_count == 3)
                           {  c_count = 0;
                              D_buff.m_error ++;  /* Flag message */
                              D_buff.p_value = 0; /* clear counters. */
                              D_buff.parity.storage = 0;
                              D_buff.move_flag = 0; }
                        else
                           {  Rechead.R_head.errind =
                                Rechead.R_head.errind | SHORTR;
                              D_buff.s_error ++;      /* short error */
                              fin_rec (temp, inp.word, c_count);
                              for (; c_count < D_buff.r_c_count;
                                     c_count += 2)
                                 {  *(D_buff.Outpoint) = 0x00;
                                    D_buff.Outpoint++; }
                              c_count = 0;
                              r_pnt ();
                           }
                     }
               }
```

```
/*              If output buffer full, empty it and reset pointers.
/*              First, pad end with 0x00 and fill header
  */

              if (Datahead.header.ncas >= Datahead.header.rec_tape)
                 { for (; D_buff.Outpoint < D_buff.Out_end;
                          D_buff.Outpoint++)
                     *(D_buff.Outpoint) = 0x00;

                   memmove (D_buff.o_point[D_buff.which_out],
                          Datahead.string, 20);
                   istat = write (fh_d ,
                          D_buff.o_point[D_buff.which_out], OUTB);
                   if (istat != OUTB)
                      { e_line ();
                        cprintf ("%s%s%d", Modul, Err6, istat);
                        e_line ();
                        cputs (Err4);
                        Datahead.header.errind = istat; }
                   else
                      Datahead.header.errind = 0;
                   if (D_buff.which_out == 0)
                      {  D_buff.which_out = 1;  }
                   else
                      {  D_buff.which_out = 0;  }

                   D_buff.Outpoint = D_buff.o_point[D_buff.which_out] +
                        13 * sizeof (short);
                   D_buff.rec_p = D_buff.o_point[D_buff.which_out] +
                        10 * sizeof (short);
                   D_buff.Out_end =D_buff.o_point[D_buff.which_out] +
                        OUTB * sizeof (char);

                   Datahead.header.nrec ++;
                   Datahead.header.ncas = 0;
                 }

/*              Check for the end of the input buffer.  If found then,
/*              reset pointers and go back for more data.
  */

              if (D_buff.Procpoint >= D_buff.Proc_end)
                 {  end_buff ();
                    D_buff.i_count = D_buff.R_count; }
           }
        }
```



```
/*      All finished with the processing of this tape (the user typed ESC).
/*      Now we need to clean this up by finishing the processing on the
/*      last few buffers and then make sure everything is out to the file.
  */
        end_p (fh_d, c_count);          /* Clean up extra buffers. */

        for (; D_buff.Outpoint < D_buff.Out_end; D_buff.Outpoint++)
               *(D_buff.Outpoint) = 0x00;

        memmove (D_buff.o_point[D_buff.which_out], Datahead.string, 20);

        istat = write (fh_d , D_buff.o_point[D_buff.which_out], OUTB);
        if (istat != OUTB)
           {  e_line ();
              cprintf ("%s%s%d", Modul, Err6, istat);
           }

        return (0);
}
```

```
/*      fin_rec
/*      Routine to finish processing the input tape record.
/*
/*      Parameters:
/*              temp    - Integer count of the number of characters in
/*                        the word being processed.
/*              word    - The word being processed.
/*              c_count - Count of characters in the entire record.
/*      Return values:
/*              none
  */

void    fin_rec (int temp, short int word, int c_count)
{
        void    e_line ();

        extern  BUFF    D_buff;         /* Storage for data buffers */
        extern  D_Head  Datahead;       /* Header area for output */
        extern  C_Head  Comhead;        /* Comment headers. */
        extern  C_field Comment;        /* Comment array. */
        extern  R_Head  Rechead;        /* Headers. */

        static  char    Modul[]={"fin_rec  --  "};
        static  char    Err5[]=
                  {"Error in record character count [>3 or <1]."};

/*              Set flag for finishing the parity calculations. */
        D_buff.p_flag = 1;

/*              Process end-of-record based on the number of
/*              characters to be moved from the input record */
        switch (temp) {
               case 1:                  /* only one character */
                        char_1 (word);
                        break;
               case 2:                  /* two characters */
                        char_2 (word);
                        break;
               case 3:                  /* three characters */
                        char_3 (word, 1);
                        break;
               default:                 /* error condition */
                        e_line ();
                        cprintf ("%s%s", Modul, Err5);
               }
        D_buff.p_flag = 0;      /* Clear parity flag */
```



```
/*              If there is a parity error, compliment the bits so
/*              they show up easily in the error word.  Check to
/*              be sure no other errors have been processed
  */
        if (Rechead.R_head.errind == 0)
           {  if (D_buff.p_stat != 0)
              { Rechead.R_head.errind =
                  Rechead.R_head.errind | ((~D_buff.p_value) & PARITY);
                D_buff.p_error ++; }     /* count the parity error */
           }
        Rechead.R_head.nchar = c_count;  /* chars. in rec. */
        return;
}


/*      r_pnt
/*      Routine to copy record header to output and reset pointers.
/*
/*      Parameters:
/*              none
/*      Return values:
/*              none.
  */

void    r_pnt ()
{
        int     i;

/*              Copy record header to output and get ready for
/*              next record.
  */
        memmove (D_buff.rec_p, Rechead.string, 6);

/*              Pad the record to an even number of words.  */

        for (i = 0; i < D_buff.pad_count; i++, D_buff.Outpoint++)
             *(D_buff.Outpoint) = 0x00;

        Datahead.header.ncas ++;        /* number of cassette rec. */
        D_buff.records ++;              /* count the records read. */

        Rechead.R_head.cas_no++;        /* number of next cass. rec. */
        Rechead.R_head.errind = 0;      /* clear error indicator. */
```

```
/*              Check the number of records and print number if
/*              an even multiple of 3000.
  */
        if (!(D_buff.records % 3000) || (D_buff.records == 1000))
           {  window (1, 18, 80, 21);
              textattr (D_buff.err_attr);
              gotoxy (19, 1);
              cprintf ("%6lu", D_buff.records);
              gotoxy (1, 4);
              cprintf
                ("   %5lu       %5lu            %5lu           %5lu",
                     D_buff.p_error, D_buff.l_error, D_buff.s_error,
                     D_buff.e_error);
           }

/*              If this is really end of a record, there should be
/*              message word in next two bytes.  Remove them and,
/*              throw them away.
  */
        if ((*(D_buff.Procpoint + 1) & 0x08) == 0x08)
           {  D_buff.Procpoint ++;       /* skip message word */
              D_buff.Procpoint ++;
              D_buff.l_count += 2;
           }

        D_buff.p_value = 0;
        D_buff.parity.storage = 0;  /* clear parity for next pass. */
        D_buff.rec_p = D_buff.Outpoint;          /* Point to next */
        D_buff.Outpoint += 3 * sizeof (short);  /* output locations */
        D_buff.move_flag = 0;
}
```



```
/*      l_rec
/*      Routine to handle a long record.  This includes copying
/*      of data to work area, counting bytes, checking for
/*      end of input buffer, and aborting program if rec. is
/*      too long.
/*
/*      Parameters:
/*              none.
/*      Return values:
/*              1 - if end-of-record found within 200 bytes.
/*              11 - if end-of-record was not found  --  failure.
  */

int     l_rec ()
{

        extern  BUFF    D_buff;
        extern  D_Head  Datahead;
        extern  R_Head  Rechead;

        void    end_buff();
        int     i, j;

        static  char    Modul[]={"l_rec  --  "};
        static  char    Err1[]=
                  {"Unable to find end of long record (> 300 char.)."};
        static  char    Err2[]={"The tape reading will abort."};

        union   {
                struct  {                       /* bytes in data */
                        unsigned char s1;       /* high byte */
                        unsigned char s2;       /* low byte */
                        } b;
                short   int  t;                 /* data word */
                } tmp;

/*              Flag the long record.           */
        Rechead.R_head.errind = Rechead.R_head.errind | LONGR;
        D_buff.l_error ++;
```

```
/*              Read in bytes of data and check for end of record.
/*              Loop through this for a max. of 200 bytes.
  */
        for (i = 0; i < 200; i += 2)
           {  tmp.b.s2 = *(D_buff.Procpoint)++;
              tmp.b.s1 = *(D_buff.Procpoint)++;
              D_buff.i_count += 2;
              Rechead.R_head.nchar += 3;

              if ((tmp.b.s1 & END) == END)       /* If end found, */
                 {   return (1);          /* set status and return. */
                 }

/*              We may find the end of the input buffer.  If so
/*              we will need to end one buffer and go wait for the
/*              next.
  */
              if (D_buff.Procpoint >= D_buff.Proc_end)
                 {  end_buff ();
                    while (D_buff.inb_count <= 0)
                       {  c_break ();
                       }
                 }
           }

/*              If we end this loop, then we have checked 200 bytes
/*              (approx. 300 char.) and not found the end of this
/*              record.  This is a hopeless situation and we need
/*              to abort the data collection.
  */
        e_line ();
        cprintf ("%s%s", Modul, Err1);
        e_line ();
        cputs (Err2);
        return (11);
}
```



```
/*      end_buff
/*      Routine to switch buffer pointers to another buffer.
/*
/*      Parameters:
/*              none.
/*      Return values:
/*              none.
  */

void    end_buff ()
{
        extern  BUFF    D_buff;         /* Data buffers */

/*              Decide which buffer to process next and increment
/*              counter to buffer or set to 0.
  */
        if (D_buff.which_proc <= 1)
           {  D_buff.which_proc++;  }
        else
           {  D_buff.which_proc = 0;  }

/*              Set pointer to buffer and end of buffer.
  */
        D_buff.Procpoint = D_buff.r_point[D_buff.which_proc];
        D_buff.Proc_end = D_buff.Procpoint + D_buff.R_count;
        return;
}
```

```
#include <conio.h>
#include <stdio.h>
#include <dos.h>
#include "seadata.h"

/*      intr_init.c
/*      T.W. Danforth           15-Feb-1990
/*      Routine which is used to initialize interrupts for
/*      the Seadata cassette reading program.
/*
/*      Parameters:
/*              none.
/*
/*      Return values:
/*              none.
  */

void    intr_init ()

{
        extern  BUFF    D_buff;         /* Data buffers & pointers */
        extern  D_Head  Datahead;       /* Data buffers headers. */
        extern  C_Head  Comhead;        /* Comment headers. */
        extern  C_field Comment;        /* Comment array. */
        extern  R_Head  Rechead;        /* Headers. */

        void    interrupt Sea_int();    /* Our interrupt handler. */

        unsigned char i_mask;

/*              Save the COM2 interrupt vector and then init. the
/*              8255 processor and set up the vector to handle the
/*              incomming data.
  */
/*               Save COM2 interrupt vector. */
        D_buff.oldfunc = getvect (COM2);
/*               Our interrupt handler. */
        setvect (COM2, Sea_int);
/*               Initialize to Mode 1, input */
        outportb (CONTROL, 0xBE);
/*               Set Port A interrupt enable.  This is the
/*               only interrupt needed since both A & B
/*               will happen together and the only line
/*               tied to the AT bus is from Port A.
  */
        outportb (CONTROL, 0x09);
```



```
/*              Disable Real-time clock interrupt - IRQ 0 - since
/*              it may interfere with data collection.  And
/*              enable interrupt for data collection on IRQ 3.
  */
/*               Get mask and disable IRQ 0 */
        i_mask = inportb (0x21) | 0x01;
/*               Set mask with ITQ 3 enabled */
        outportb (0x21, (i_mask & 0xF7));

        D_buff.mes_line = 0;
        m_line ();
        clrscr ();

        cputs ("Tape reading initialized.");
        m_line ();
        cputs ("You may start the reader at any time.");
        m_line ();
        cputs ("To stop the reading:     Stop the reader, then");
        m_line ();
        cputs ("                         press the ESC key.");

        return;
}
```

```
#include <unixio.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <file.h>

#define VERSION 1.00            /* Version of the software */
#define OUTB    8192            /* Size of output buffers. = 8192 */

/*      pcarp.c
/*      T.W. Danforth           28-Mar-1990
/*
/*      Program to reformat carp data files created on a PC using
/*      the seadata program.  The input is two files which have names
/*      ending with .CMM and .DAT for comment and data respectively.
/*      These files are read and converted to CRAP format (LSI).  The
/*      resultant data will appear to the CARPBIT program to have been
/*      read on the LSI.  PCARP must be run on the VAX to convert data
/*      files on disk.
/*
/*      Parameters:
/*              file    - the root name of the input data files.  This
/*                        name is passed on the command line and must
/*                        specify the root name of the files to be
/*                        converted.  The program searches this string for
/*                        a period (".") and deletes anything beyond the
/*                        first period found, assuming that this is the
/*                        file name extension.  Any directory or logical
/*                        name information must, therefore, not contain a
/*                        period.  The file extensions .CMM and .DAT will
/*                        be added by the program for the input file names.
/*                        The output file will have the same name with the
/*                        extension of .LSI.  The input and output files
/*                        will be in the same directory.  If the output
/*                        file already exists, it will be zeroed.
/*
/*                        If the parameter is not specified, the
/*                        program will go into interactive mode and prompt
/*                        for the parameter.
  */
main (argc, argv)
        int     argc;
        char    *argv[];
{
```



```
        extern  int     errno;          /* error number. */

/*      Data header
/*      Structure used to initialize output data records.
/*      This gets copied over the start of the output buffer.
  */
struct data {
        unsigned short  FMT;            /* Record indicator - 0x4A4E */
        unsigned short  char_rec;       /* # char/rec. */
        unsigned short  rec_tape;       /* # cassette rec./output rec. */
        unsigned short  wrd_rec;        /* # words/cassette rec. */
                                        /* Equal to char_rec / 4 + 3.75 */
        unsigned short  nrec;           /* Sequential tape record #.  */
        unsigned short  ncas;           /* Number of cassette records*/
                                        /*   in output record. */
        unsigned short  errind;         /* Error indicator.  Non-zero = */
                                        /*   write error on prev. rec. */
        unsigned short  dumb[3];
        } ;
union   { char          string[20];
          struct        data    head;
        } Datahead;

/*      Comment header.
/*      Structure which builds the header for a comment block.
  */
struct  {
        unsigned short  COMFMT;         /* Comment indicator - 0x0FFA */
        unsigned short  data_fmt;       /* format for data field */
        unsigned short  char_rec;       /* # char. / record */
        unsigned short  rec_tape;       /* cassette rec. / output rec. */
        unsigned short  wrd_rec;        /* words/cassette rec. */
        unsigned short  flag;           /* archaic flag for 9-track tape */
        unsigned short  irtype;         /* reader type indicator */
        unsigned short  comp;           /* computer used for reading */
        unsigned short  b_len;          /* length of the output buffers */
        unsigned short  dumb[3];
        } Comment;
```

```
void    read_err (int);
void    writ_err (int);
void    bswap (char *, char *, int);

int     fh_c, fh_d, fh_l;       /* file handles for cmm & dat */
int     jj, kk, i, l;
short int hwrd, cassette;

unsigned char   *p, froot[40];  /* file name and pointer */
/*              input and output buffers and their pointers. */
unsigned char   buff[8192], obuff[8192];
unsigned char   *ip, *op, *out_end;
unsigned char   temp[30];

out_end = &obuff[0] + OUTB * sizeof (unsigned char);
for (i = 0; i < 3; i++)
    Comment.dumb[i] = 0;

if (argc <= 1)
   {  printf ("File name: ");   /* No file on command line. */
      if (gets (froot) == NULL)
          exit (EIO);
   }
else
   {  strcpy (froot, argv[1]);  /* copy param. to froot */ }


/*              Scan for a '.' and, if not found, then add '.DAT' and
/*              '.CMM' for the files.
  */
p = strchr (froot, '.');
if (p == 0)                     /* Point to place for file extension */
    p = &froot[0] + strlen (froot);
```



```
/*              Add the '.DAT' or the '.CMM' and then call
/*              the open routine.
  */
strcpy (p, ".CMM");             /* build the comment file name */
fh_c = open (froot, O_RDONLY, 0);       /* open comment file */
if (fh_c == -1)
   {  printf ("Error opening comment file %s\n", froot);
      printf ("Error: %d\nProgram exiting\n", errno);
      exit (EIO); }

strcpy (p, ".DAT");             /* build the data file name */
fh_d = open (froot, O_RDONLY, 0);       /* open data file */
if (fh_d == -1)
   {  printf ("Error opening data file %s\n", froot);
      printf ("Error: %d\nProgram exiting\n", errno);
      exit (EIO); }

strcpy (p, ".LSI");
fh_l = creat (froot, 0, "ctx = rec",    /* open output file */
                "mrs = 8192", "rfm = var");
if (fh_d == -1)
   {  printf ("Error opening output file %s\n", froot);
      printf ("Error: %d\nProgram exiting\n", errno);
      exit (EIO); }

/*              Read the comment file and reformat the file header.
/*              Write the header and then copy the remains of the
/*              comment file to the output.
  */
jj = read (fh_c, buff, 51);
if (jj != 51)
    read_err (jj);              /* handle read error */

jj = sscanf (buff,
        "%4hx %4hx %3hd %5hd %4hd %3hd %3hd %3hd %5hd\n",
        &Comment.COMFMT, &Comment.data_fmt,
        &Comment.char_rec, &Comment.rec_tape,
        &Comment.wrd_rec, &Comment.flag,
        &Comment.irtype, &Comment.comp, &Comment.b_len);
if (jj != 9)
   {  printf ("Comment data scan error\nProgram exiting\n");
      exit (4); }
```

```
/*              Check the comment field and swap if necessary, also set
/*              the computer type field to look like the proper machine.
  */
        Comment.comp = -1;                  /* fix for the LSI */

        jj = write (fh_1, &Comment, 24);          /* Write header */
        if (jj != 24)
            writ_err (jj);                  /* handle write error */

/*              Loop until the end of the comment file and copy the
/*              comments to output.
  */
        memmove (&buff[0], &Comment.COMFMT, 2);
        jj = 1;
        while (jj != 0)
            {   jj = read (fh_c, &buff[2], 71);
                if (jj == 0)                    /* end of file */
                    break;
                if (jj == -1)
                    read_err (jj);              /* handle read error */
                kk = write (fh_1, buff, 70);  /* write what we read. */
                if (kk != 70)
                    writ_err (kk);
            }

/*              Loop to read the data file.  Read 8192 bytes at a time
/*              and then reformat the data.  The headers for both the
/*              buffer and cassette record have the bytes in the proper
/*              order.  The actual data, however, needs to have the
/*              bytes swapped.
/*              The end of the loop is an end-of-file.
  */
        jj = 1;
        while (jj != 0)
```



```
/*              Loop to read the blocks of data.  Need to read
/*              16 blocks of data to get one PC-Carp data record.
  */
            {   ip = &buff[0];
                for (i = 0; i < 16; i++, ip += 512)
                    {   jj = read (fh_d, ip, 512);
                        if (jj == 0)
                            goto finish;
                        if (jj != 512)
                            read_err (jj);
                    }

                memmove (Datahead.string, &buff[0], 20);  /* LSI */
                cassette = Datahead.head.ncas;    /* records in block */
                nwrd = Datahead.head.wrd_rec - 3; /* words / rec.  */
                memmove (&obuff[0], Datahead.string, 20);

                op = &obuff[20];  /* Point to output location */
                ip = &buff[20];

/*                Copy individual cassette records.  If LSI, headers get
/*                moved without any changes and the data gets moved while
/*                swapping the bytes.
/*                              LSI - move header and swap data.  */
                for (i = 0; i < cassette; i++)
                    {   for (i = 1; i <= 6; i++, ip++, op++)
                            *op = *ip;
                        bswap (ip, op, nwrd);
                        ip += nwrd * 2;     /* increment pointer */
                        op += nwrd * 2;
                    }
/*                Finish the output buffer by filling with nulls.
  */
                for (; op <= out_end; op++)
                    *op = 0;
/*                Write the buffer to the output file.  */
                kk = write (fh_1, obuff, OUTB);
                if (kk != OUTB)    writ_err (kk);
            }
```

```
/*              Close the files.  */
finish:
        close (fh_c);
        close (fh_d);
        close (fh_l);
}

/*              Routine to swap bytes */

void    bswap (char *ip, char *op, int nwrd)
{
        int     i;

        for (i = 1; i <= nwrd; i++)
           {  *op = *(ip + 1);  /* next byte into output */
              op ++;
              *op = *ip;                /* this byte into next output */
              ip += 2;
              op ++; }
}
/*              Routines to process i/o errors.
/*              All error messages are printed and then
/*              the program quits from here.
  */
void    read_err (int jj)
{
        printf ("Error during read  --  %d bytes read\n", jj);
        perror ("pcarp");
        exit (errno);
}

void    writ_err (int jj)
{
        printf ("Error during write  --  %d bytes written\n", jj);
        perror ("pcarp");
        exit (errno);
}
```

```
#include <stdlib>
#include <unixio.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <file.h>

#define VERSION 1.00            /* Version of the software */
#define OUTB    1600            /* Size of output buffers. = 1600 */
#define INB     8192            /* Size of input buffer. */

/*      pcarphp.c
/*      T.W. Danforth           28-Mar-1990
/*
/*      Program to reformat carp data files created on a PC using
/*      the seadata program.  The input is two files which have names
/*      ending with .CMM and .DAT for comment and data respectively.
/*      These files are read and converted to CARP format (HP).
/*      The resultant data file will appear to CARPBIT and other programs
/*      to have been read on the HP.
/*      PCARP must be run on the VAX to convert data files on disk.
/*
/*      Parameters:
/*              file    - the root name of the input data files.  This
/*                        name is passed on the command line and must
/*                        specify the root name of the files to be
/*                        converted.  The program searches this string for
/*                        a period (".") and deletes anything beyond the
/*                        first period found, assuming that this is the
/*                        file name extension.  Any directory or logical
/*                        name information must, therefore, not contain a
/*                        period.  The file extensions .CMM and .DAT will
/*                        be added by the program for the input file names.
/*                        The output file will have the same name with
/*                        the extension of .HP.  The input and output files
/*                        will be in the same directory.  If the output file
/*                        already exists, a new version will be created.
```



```
/*
/*              Access  - the route via which the data reached the VAX.
/*                        This will be either a 9-track tape written on
/*                        the PC or ethernet.  Since the most common at
/*                        the time the program was written is tape, this
/*                        will be the default.  The user must enter "tape"
/*                        or "enet" on the command line.  If not specified
/*                        and the file name is specified, this will default
/*                        to "tape".  The reason for the difference is that
/*                        the VAX handles files from the two sources
/*                        differently and, thus, pcarphp must handle them
/*                        differently.
/*
/*                        If the parameters are not specified, the
/*                        program will go into interactive mode and prompt
/*                        for the parameters.
  */

main (argc, argv)
        int     argc;
        char    *argv[];

{
        extern  int     errno;          /* error number. */

/*      Data header
/*      Structure used to initialize output data records.
/*      This gets copied over the start of the output buffer.
  */
struct data {
        unsigned short  FMT;            /* Data indicator - 0x4A4E */
        unsigned short  char_rec;       /* # char/rec. */
        unsigned short  rec_tape;       /* # cassette rec./output rec. */
        unsigned short  wrd_rec;        /* # words/cassette rec. */
                                        /* Equal to char_rec / 4 + 3.75 */
        unsigned short  nrec;           /* Sequential tape record #.  */
        unsigned short  ncas;           /* Number of cassette records in */
                                        /*   this output record. */
        unsigned short  errind;         /* Error indicator.  Non-zero = */
                                        /*   write error on prev. */
                                        /*   output record.  */
        unsigned short  dumb[3];
        } ;
union   {
        char    string[20];
        struct  data    head;
        } Datahead;
```

```
/*      Comment header.
/*      Structure which builds the header for a comment block.
  */
struct {
        unsigned short  COMFMT;         /* Comment indicator - 0x0FFA */
        unsigned short  data_fmt;       /* format for data field */
        unsigned short  char_rec;       /* # char. / record */
        unsigned short  rec_tape;       /* cassette rec. / output rec. */
        unsigned short  wrd_rec;        /* words/cassette rec. */
        unsigned short  flag;           /* archaic flag for 9-track tape */
        unsigned short  irtype;         /* reader type indicator */
        unsigned short  comp;           /* computer used for reading */
        unsigned short  b_len;          /* length of the output buffers */
        unsigned short  dumb[3];
        }  Comment;


        void    read_err (int);
        void    writ_err (int);
        void    bswap (char *, char *, int);

        int     fh_c, fh_d, fh_l;       /* file handles for cmm & dat */
        int     jj, kk, i, l, first_r, second, d_read;

        short int tmp1, tmp2;
        short int hp_rec, hp_max, init = 0, out_rec;
        short int nwrd, cassette;

        char    Access[6];
        unsigned char   *p, froot[40];  /* file name and pointer */
/*              Input and output buffers and pointers to them. */
        unsigned char   buff[INB], obuff[OUTB];
        unsigned char   *ip, *op, *out_end;
        unsigned char   temp[30];
```



```
        out_end = &obuff[0] + OUTB * sizeof (unsigned char);
        for (i = 0; i < 3; i++)
            Comment.dumb[i] = 0;

        if (argc <= 1)
           {  printf ("File name: ");   /* No file on command line. */
              if (gets (froot) == NULL)
                 exit (EIO);
              printf ("\nAccess method: ");
              if (gets (Access) == NULL)
                 exit (EIO);
              printf ("\n");
           }
        else
           {  strcpy (froot, argv[1]);  /* copy com-line param. to froot */
              if (argc <= 2)
                  strcpy (Access, "TAPE");      /* Access type string */
              else
                  strcpy (Access, argv[2]);
           }

/*              Upcase the access string and then set comment read
/*              parameters based on this string.
  */
        for (i = 0; i < strlen (Access); i++)
            Access[i] = toupper (Access[i]);
        if (strncmp (Access, "TAPE", 4) == 0)
           {  first_r = 52;             /* For tape access, read the  */
              second = 72;              /* extra bytes of carriage    */
              d_read = 513;             /* control and discard.       */
           }
        else
           {  first_r = 51;             /* Ethernet access parameters */
              second = 71;
              d_read = 512;
           }
```

```
/*              Scan for a '.' and, if not found, then add '.DAT' and
/*              '.CMM' for the files.
 */
        p = strchr (froot, '.');
        if (p == 0)                 /* Point to place for file extension */
            p = &froot[0  + strlen (froot);

 *              Add the '.DAT' or the '.CMM' and then call
 *              the open routine.
 */
        strcpy (p, ".CMM");                 /* build comment file name */
        fh_c = open (froot, O_RDONLY, 0);       /* open comment file */
        if (fh_c == -1)
            {  printf ("Error opening comment file %s\n", froot);
               printf ("Error: %d\nProgram exiting\n", errno);
               exit (EIO);
            }

        strcpy (p, ".DAT");                 /* build the data file name */
        fh_d = open (froot, O_RDONLY, 0);       /* open data file */
        if (fh_d == -1)
            {  printf ("Error opening data file %s\n", froot);
               printf ("Error: %d\nProgram exiting\n", errno);
               exit (EIO);
            }

        strcpy (p, ".HP");

        fh_l = creat (froot, 0, "ctx = rec",    /* open output file */
                      "mrs = 1600", "rfm = var");
        if (fh_d == -1)
            {  printf ("Error opening output file %s\n", froot);
               printf ("Error: %d\nProgram exiting\n", errno);
               exit (EIO);
            }

/*              Read the comment file and reformat the file header.
/*              Write the header and then copy the remains of the
/*              comment file to the output.
 */
        jj = read (fh_c, buff, first_r);
        if (jj != first_r)
            read_err (jj);                  /* handle read error */
```



```
        jj = sscanf (buff, "%4hx %4hx %3hd %5hd %4hd %3hd %3hd %3hd %5hd",
               &Comment.COMFMT, &Comment.data_fmt,
               &Comment.char_rec, &Comment.rec_tape,
               &Comment.wrd_rec, &Comment.flag,
               &Comment.irtype, &Comment.comp, &Comment.b_len);
        if (jj != 9)
            {  printf ("Comment data scan error\nProgram exiting\n");
               exit (4);
            }
/*              Set some HP specific parameters and then swap the bytes
/*              in the comment field.
 */
        nwrd = Comment.wrd_rec - 3;     /* 8 words / rec. */
        hp_max = ((OUTB / 2) / Comment.wrd_rec) - 1; /* recs / out buff */
        Comment.rec_tape = hp_max;
        Comment.b_len = 0;              /* don't need for HP style */

        Comment.data_fmt = 0x3031;
        Comment.comp = 0;
        bswap (&Comment, temp, 12);     /* swap for HP */
        memmove (&Comment, temp, 24);

        jj = write (fh_l, &Comment, 24);        /* Write comment header */
        if (jj != 24)
            writ_err (jj);              /* hand.e write error */

/*              Loop until the end of the comment file and copy the
/*              comments to output.
 */
        memmove (&buff[0], &Comment.COMFMT, 2);
        jj = 1;
        while (jj != 0)
            {  jj = read (fh_c, &buff[2], second);
               if (jj == 0)             /* end of file */
                   break;
               if (jj == -1)
                   read_err (jj);       /* handle read error */
               kk = write (fh_l, buff, 70);  /* write what we read. */
               if (kk != 70)
                   writ_err (kk);
            }
```

```
/*              Loop to read the data file.  Read 8192 bytes at a time
 *              and then reformat the data.  The headers for both the
 *              buffer and cassette record have the bytes in the proper
 *              order.  The actual data, however, needs to have the
 *              bytes swapped.
 *              The end of the loop is an end-of-file.
 */

        out_rec = 0;
        hp_rec = 0;
        jj = 1;
        while (jj != 0)

/*              Loop to read the blocks of data.  Need to read
 *              16 blocks of data to get one PC-Carp data record.
 */

            {  ip = &buff[0];
               for (i = 0; i < 16; i++, ip += 512)
                  {  jj = read (fh_d, ip, d_read);
                     if (jj == 0)
                        goto finish;
                     if (jj != d_read)
                        read_err (jj);
                  }

/*              Get number of records in input buffer and the number of
 *              words/record.  Use for copying and swapping the bytes.
 *              Write the data header to the output buffer.
 */

               memmove (&cassette, &buff[10], 2);

               bswap (&buff[0], Datahead.string, 10);    /* HP  */
               Datahead.head.rec_tape = Comment.rec_tape;
               ip = &buff[20];

               if (init == 0)    /* copy to output buff if first */
                  {  memmove (&obuff[0], Datahead.string, 20);
                     out_rec = 0;
                     init ++;
                     op = &obuff[20];    /* Point to output location */
                  }
```



```
/*              Copy the individual cassette records.  The headers get
 *              swapped, and the data gets moved as is.
 *              Loop based on number of records in input data buffer.
 *              The output is based on an inner loop.
 */

/*                          HP - swap header and move data.  */
               for (i = 0; i < cassette; i++)
                  {  bswap (ip, op, 3);
                     ip += 6;             /* increment pointer */
                     op += 6;
                     for (i = 1; i <= nwrd * 2; i++, ip++, op++)
                         *op = *ip;

                     hp_rec ++;           /* increment output count */
                     if (hp_rec == hp_max)
                        {  for (; op <= out_end; op++)  /* finish output */
                               *op = 0;
                           out_rec ++;
                           bswap (&out_rec, &tmp2, 1);
                           Datahead.head.nrec = tmp2;

                           bswap (&hp_rec, &tmp2, 1);
                           Datahead.head.ncas = tmp2;
                           Datahead.head.FMT = Comment.data_fmt;
                           hp_rec = 0;

                           memmove (&obuff[0], Datahead.string, 20);
                           op = &obuff[20];      /* Point to out location */

                           kk = write (fh_1, obuff, OUTB);
                           if (kk != OUTB)
                               writ_err (kk);
                        }
                  }
            }
```

```
finish:
/*              Finish the last output buffer by filling with nulls.
  */
        for (; op <= out_end; op++)
            *op = 0;

/*              Write the buffer to the output file.  */

        out_rec ++;
        bswap (&out_rec, &tmp2, 1);
        Datahead.head.nrec = tmp2;

        bswap (&np_rec, &tmp2, 1);
        Datahead.head.ncas = tmp2;
        memmove (&obuff[0], Datahead.string, 20);

        kk = write (fh_1, obuff, OUTB);
        if (kk != OUTB)
           writ_err (kk);

/*              Close the files.  */
        close (fh_c);
        close (fh_d);
        close (fh_1);
}


/*              Routine to swap bytes in a series of bytes.
  */

void    bswap (char *ip, char *op, int nwrd)
{
        int     i;

        for (i = 1; i <= nwrd; i++)
           {  *op = *(ip + 1);  /* next byte into output */
              op ++;
              *op = *ip;        /* this byte into next output */
              ip += 2;
              ip ++;
           }
}
```



```
/*              Routines to process i/o errors.
/*              All error messages are printed and then
/*              the program quits from here.
  */

void    read_err (int jj)
{
        printf ("Error during read  --  %d bytes read\n", jj);
        perror ("pcarp");
        exit (errno);
}


void    writ_err (int jj)
{
        printf ("Error during write  --  %d bytes written\n", jj);
        perror ("pcarp");
        exit (errno);
}
```

```
#include <conio.h>
#include <stdio.h>
#include "seadata.h"

/*      screen.c
/*      T.W. Danforth           15-Feb-1990
/*      Routines to print error counts in the error window on the
/*      screen.  Used with the PC Carp program.
/*
/*      e_line
/*      Routine to position lines in the error window.
  */
void    e_line  ()
{
        extern  BUFF    D_buff;

        if (D_buff.er_line == 4)
            D_buff.er_line = 1;         /* reset to the top. */
        else
            D_buff.er_line ++;          /* go to next line */

        window (1, 22, 80, 25);
        textattr (D_buff.e_attr);
        gotoxy (1, D_buff.er_line);
        clreol ();
        return;
}

/*      m_line
/*      Routine to position message lines.
  */
void    m_line  ()
{
        extern  BUFF    D_buff;

        if (D_buff.mes_line == 6)
            D_buff.mes_line = 1;        /* reset to the top. */
        else
            D_buff.mes_line ++;         /* go to next line */

        window (1, 12, 80, 17);
        textattr (D_buff.m_attr);
        gotoxy (1, D_buff.mes_line);
        clreol ();
        return;
}
```



```
/*      p_line
/*      Routine to position prompt lines.
  */
void    p_line  ()
{
        extern  BUFF    D_buff;

        if (D_buff.prm_line >= 8)
            D_buff.prm_line = 1;        /* reset to the top. */
        else
            D_buff.prm_line ++;         /* go to next line */

        window (1, 4, 80, 11);
        textattr (D_buff.p_attr);
        gotoxy (1, D_buff.prm_line);
        clreol ();
        return;
}

/*      t_line
/*      Routine for permanent lines in the top window.
  */
void    t_line  ()
{
        extern  BUFF    D_buff;

        if (D_buff.top_line >= 3)
            D_buff.top_line = 2;        /* reset to the top. */
        else
            D_buff.top_line ++;         /* go to next line */

        window (1, 1, 80, 3);
        textattr (D_buff.topw);
        gotoxy (1, D_buff.top_line);
        clreol ();
        return;
}
```

```
#include <math.h>
#include <stdio.h>
#include "seadata.h"
#include <errno.h>
#include <conio.h>

/*      seacom.c
/*      T.W. Danforth           15-Aug-1989
/*      Function to get information about the tape being processed,
/*      build the data headers (initialize with the information the
/*      user enters), and comments about the seadata cassette.
/*      Parameters:
/*              fh  - The file handle of the output file.
/*
/*      Returns:
/*              1 = okay.
/*              -3 = fatal write error.
/*              -5 = fatal error reading input.
  */

int     seacom (int fh)
{

        extern  BUFF    D_buff;         /* Data buffers. */
        extern  D_Head  Datahead;       /* Data record headers. */
        extern  C_Head  Comhead;        /* Comment header. */
        extern  R_Head  Rechead;        /* Output buffer header. */
        extern  C_field Comment;        /* Comment storage */

        extern  int     errno;
        extern  char    *sys_errlist[];

        static  char    Err1[]=
                {"The number of characters in the cassette record."};
        static  char    Err2[]={"must be >= 1 and < 256."};
        static  char    Err4[]={"Error on write to disk file."};

        void    read_com(int, char *);
        void    p_line(void), e_line(void), m_line(void);
        char    chead[80], str[6];
        int     i, jj, istat;

        str[0] = 4;
```



```
/*      Get basic information about the tape and experiment.
/*      Start with the characters/cassette record and then
/*      calculate some other parameters.
  */

        istat = 0;

        while (istat != 1)
        {  D_buff.prm_line = 6;
           p_line ();
           cprintf
        ("Enter the number of characters/cassette record (1-255): ");
           cgets (str);
/*               Convert string to integer */
           Datahead.header.char_rec = atoi (&str[2]);

/*              Check to be sure the conversion worked and that
/*              the number is within limits.
  */

           if (Datahead.header.char_rec <= 0 ||
                Datahead.header.char_rec >=256)
              { e_line ();
                cputs (Err1);
                e_line ();
                cputs (Err2);
                continue; }
           else
                istat = 1;
        }

        Datahead.header.FMT = 0x4A4E;
        i = Datahead.header.char_rec / 4;
        Datahead.header.wrd_rec = ceil (i + 3.75);
        Datahead.header.rec_tape = (((OUTB-100) / 2) /
                Datahead.header.wrd_rec ) -1;

/*              Number of chars. to expect + parity + garbage */
        Datahead.header.char_rec += D_buff.adex;
        D_buff.r_c_count = Datahead.header.char_rec;
```

```
/*      Build the header for the comment field  --  this also
/*      is the header record for all of the data.  After
/*      building this, write it to the comment file.
  */
        Comhead.COMFMT = 0x0FFA;
        Comhead.data_fmt = Datahead.header.FMT;
        Comhead.char_rec = Datahead.header.char_rec;
        Comhead.rec_tape = Datahead.header.rec_tape;
        Comhead.wrd_rec  = Datahead.header.wrd_rec;
        Comhead.flag     = -1;
        Comhead.irtype   = D_buff.irtyp;
        Comhead.comp     = -2;  /* indicates PC  */
        Comhead.b_len    = OUTB;        /* len - output buffer */
        for (jj = 0; jj < 3; jj++)
            {  Comhead.dumb[jj] = 0;
               Datahead.header.dumb[jj] = 0; }
        sprintf (chead,
                "%4X  %4X  %3d  %5d  %4d  %3d  %3d  %3d  %5d\n",
                Comhead.COMFMT, /* comment header indicator */
                Comhead.data_fmt,  /* data record indicator */
                Comhead.char_rec,  /* characters/cassette rec */
                Comhead.rec_tape,  /* cassette recs/output rec */
                Comhead.wrd_rec,  /* words in each cassette rec */
                Comhead.flag,   /* archaic flag */
                Comhead.irtype, /* reader type */
                Comhead.comp,   /* indicates PC  */
                Comhead.b_len); /* length of output buffers. */

        jj = strlen (chead);
        istat = write (fh, &chead, jj);
        if (istat != jj)
           { D_buff.er_line = 4;
             e_line ();
             cputs (Err4);
             e_line ();
             cputs (sys_errlist[errno]);
             return (-3); }

/*              Get the comment lines and save them in the
/*              comment array.
  */
        Comment.C_num = 0;
        read_com(0, chead);
        return (1);
}
```



```
/*      read_com
/*      Routine to read comment lines into an array.  This cam be
/*      used both before and after the tape has been read.
/*      Most of the information is kept in the structure C_field.
/*
/*      Parameters:
/*              flg     - 0 = first pass,
/*                            don't build summary comment.
/*                        1 = second pass,
/*                            build summary comment.
/*              time    - Pointer to string containing the time.
/*                            This is only used during the second
/*                            pass and can be a dummy pointer
/*                            during first call.
/*      Returns:
/*              none.
  */

void    read_com(int flg, char *time)
{
        extern  BUFF    D_buff;
        extern  C_field Comment;        /* Comment storage */

        int     k, flushall ();
        char    *pnt, *gets(char *);
        void    p_line (void), m_line (void);

        static  char *words[15]={"fifteen", "fourteen", "thirteen",
                        "twelve", "eleven", "ten", "nine", "eight",
                        "seven", "six", "five", "four", "three",
                        "two", "one"};
        char    com[73];


        com[0] = 70;            /* Length of input comment buff. */
        com[72] = 0;
```

```
/*              If second pass, build summary comment and put it
/*              into the comment array.
  */
        if (flg == 1)
           {  sprintf (Comment.C_array[Comment.C_num],
                " %6lu records processed  --  %s\0",
                D_buff.records, time);
              Comment.C_num ++;
              sprintf (Comment.C_array[Comment.C_num],
                 " Errors: %5lu parity,  %5lu long records,\0",
                        D_buff.p_error, D_buff.l_error);
              Comment.C_num ++;
              sprintf (Comment.C_array[Comment.C_num],
                 "         %5lu short records,  %5lu tape errors\0",
                        D_buff.s_error, D_buff.e_error);
              Comment.C_num ++;
           }
        p_line ();
        clrscr ();
        D_buff.prm_line = 6;
        p_line ();
        cprintf
        ("Enter comments - up to %s lines of information with a max. of",
              words[Comment.C_num]);
        p_line ();
        cputs
        ("70 characters/line.  End entry with an empty line (return only).");

/*              Loop up to 15 times to read the comments and store
/*              them into lines in the comment array.
  */
        window (1, 12, 80, 17);
        textattr (D_buff.m_attr);
        clrscr ();
        flushall ();

        k = 1;
        while (Comment.C_num < 15)
        {  gotoxy (1, k);
           cprintf ("%d> ", Comment.C_num + 1);
           clreol ();
           cgets(com);          /* Read comment string. */
```



```
/*              Check for a null string.  This will
/*              terminate the reading.
  */
           if (com[1] == 0)
               return;

/*              Not null string, copy to storage array and go
/*              get the next string.
  */
           strcpy (&Comment.C_array[Comment.C_num], &com[2]);
           Comment.C_num++;             /* increment array index */
           k++;                 /* increment screen line index */

/*              Check for the end of a screen and the need to start
/*              at the top of the screen.
  */
           if (k > 6)
               k = 1;
        }

        return;
}
```

```
/*      seadata.h
/*      T.W. Danforth           26-Sept-89
/*
/*      Common data areas for PC-Carp programs.
/*
  */

#define VERSION 1.01            /* Version of the software */

#define RAWB    2048            /* Size of Raw input buffers = 2048 */
#define OUTB    8192            /* Size of Output buffers. = 8192 */

#define LONGR   0x0100          /* Bit for long record error    */
#define SHORTR  0x0080          /* Bit for short record error   */
#define ERROR   0x1000          /* Bit for Seadata reader error */
#define PARITY  0x000F          /* Bits for parity error        */

#define END     0x08            /* Bit marking the end of the record    */
#define CHAR    0x06            /* Bits indicating the character count  */
#define ERRF    0x01            /* Bit flagging Seadata reader error    */

#define PORT_A  0x030C          /* Address of 8255 Port A       */
#define PORT_B  0x030D          /* Address of 8255 Port B       */
#define PORT_C  0x030E          /* Address of 8255 Port C       */
#define CONTROL 0x030F          /* Address of 8255 Control Port */
#define COM2    0x0B            /* Number of Com2 interrupt -   */
                                /* maps to addr. 0x30 */


/*      Pointers.
/*      Structure of pointers to the input and output buffers.
/*      Also contains some counters, common storage areas, and flags.
  */
struct  pl  {
        unsigned char   s1;             /* upper byte of data word. */
        unsigned char   s2;             /* lower byte of data word. */
        };
union   par  {
        short int       storage;        /* Parity storage accumulator. */
        struct pl       ps;             /* Upper & lower bytes */
        };
```



```
typedef struct  {
        unsigned char   *Rawpoint;      /* Current input buffer */
        unsigned char   *Outpoint;      /* Current output buffer */
        unsigned char   *Procpoint;     /* Current processing buffer */
        unsigned char   *rec_p;         /* Start of record header. */
        unsigned char   *r_point[3];    /* Pointers to input buffers. */
        unsigned char   *o_point[2];    /* Pointer to output buffers. */
        unsigned char   *Raw_end;       /* End of raw data buffer. */
        unsigned char   *Proc_end;      /* End of the proc. buffer */
        unsigned char   *Out_end;       /* End of output buffer. */

              /*    Indexes to indicate which input or output */
              /*    buffers are being filled or processed.    */
        unsigned int    which_raw;
        unsigned int    which_out;
        unsigned int    which_proc;
        unsigned short  inb_count;      /* Outstanding input buffers. */
        unsigned int    i_count;        /* Bytes processed in input */

        unsigned char   Raw1[RAWB];     /* Three input buffers. */
        unsigned char   Raw2[RAWB];
        unsigned char   Raw3[RAWB];

        unsigned char   Out1[OUTB];     /* Two output buffers. */
        unsigned char   Out2[OUTB];

        unsigned int    r_c_count;      /* Number of 4 bit characters to */
                                        /* expect in an input record.    */
        unsigned int    R_count;        /* Usable bytes in input buff */
        unsigned int    L_count;        /* Usable bytes in last buffer */
        unsigned int    pad_count;      /* Bytes to add to make even */
                                        /* number of words in output */

        unsigned short  irtyp;          /* Alternative indicator of reader */
        unsigned short  adex;           /* Indicator of reader type */
                                        /* and number of garbage char. */
                                        /* added to the record. */
        unsigned long   records;        /* Count of the records read */
                                        /* from cassette. */

                        /*      Indicator flags */
        unsigned short  move_flag;      /* = 0, move on byte boundary */
        unsigned short  p_flag;         /* = 1, finish parity calc */
        unsigned short  end_proc;       /* = 1, end processing. */

                        /*      Parity data */
        unsigned short  p_stat;         /* The status of parity calc. */
        short int       p_value;        /* The parity value. */
        union   par     parity;         /* Structure of parity data.   */
```

```
                        /*      Vector and error storage */
        unsigned long   s_error;        /* short record errors. */
        unsigned long   l_error;        /* long record errors. */
        unsigned long   p_error;        /* parity errors. */
        unsigned long   e_error;        /* error records. */
        unsigned long   m_error;        /* missed data words. */

        unsigned short  er_line;        /* position - error lines */
        unsigned short  mes_line;       /* position - message lines. */
        unsigned short  prm_line;       /* position - prompt lines. */
        unsigned short  top_line;       /* position - top window. */
        unsigned short  p_attr;         /* screen color attributes. */
        unsigned short  topw;
        unsigned short  m_attr;
        unsigned short  err_attr;
        unsigned short  e_attr;

        void interrupt (*oldfunc) ();   /* Storage for current interrupt */
                                        /* for COM2.  This is saved so it*/
                                        /* can be restored after data */
                                        /* collection. */
} BUFF;

/*      Data header
/*      Structure used to initialize output data records.
/*      This gets copied over the start of the output buffer.
  */
struct data {
        unsigned short  FMT;            /* Data record indicator - 0x4A4E */
        unsigned short  char_rec;       /* # char/rec. - entered by user. */
        unsigned short  rec_tape;       /* usual # cassette rec./output rec. */
        unsigned short  wrd_rec;        /* # words/cassette rec. */
                                        /*   Equal to char_rec / 4 + 3.75 */
        unsigned short  nrec;           /* Sequential tape record #.  */
        unsigned short  ncas;           /* Number of cassette records in */
                                        /*   this output record. */
        unsigned short  errind;         /* Error indicator.  Non-zero */
                                        /*   indicates write error on prev. */
                                        /*   output record.  */
        unsigned short  dumb[3];
        };

typedef union    {
        char    string[20];             /* Dummy string to match the header */
        struct  data    header;         /* Data header */
  } D_Head;
```



```
/*      Comment header.
/*      Structure which builds the header for a comment block.
  */
typedef struct  {
        unsigned short  COMFMT;     /* Comment record indicator - 0x0FFA */
        unsigned short  data_fmt;   /* format for data field */
        unsigned short  char_rec;   /* # char. / record */
        unsigned short  rec_tape;   /* cassette rec. / output rec. */
        unsigned short  wrd_rec;    /* words/cassette rec. */
        unsigned short  flag;       /* archaic flag for 9-track tape */
        unsigned short  irtype;     /* reader type indicator */
        unsigned short  comp;       /* computer used for reading */
        unsigned short  b_len;      /* length of the output buffers */
        unsigned short  dumb[3];
        }  C_Head;


/*      Comment array.
/*      Structure which combines the array of comments with
/*      an index into the array.
  */

typedef struct  {
        char    C_array[15][71];        /* Comment array - */
                                        /*  15 lines x 70 char. */
        int     C_num;                  /* Comment pointer. */
  } C_field;

/*      Record block.
/*      Structure used to start a cassette record in the output file.
/*      Each record is initialized with the following 3 variables.
  */
struct  recf    {
        unsigned short nchar;   /* Number of 4 bit characters found */
                                /* in the cassette record. */
        unsigned short errind;  /* Error indicator word. */
        unsigned short cas_no;  /* Cassette record number. */
        };

typedef union   {
        char    string[6];      /* Dummy string used for moving header */
        struct  recf    R_head; /* Cassette record header. */
  } R_Head;
```

```
#include <conio.h>
#include <stdio.h>
#include <time.h>
#include "seadata.h"

#include <sys\stat.h>
#include <fcntl.h>
#include <io.h>

/*      seadata.c
/*      T.W. Danforth           15-aug-1989
/*      This routine replaces the Seadata cassette reading program which
/*      written to run on an LSI-11.  The data is read from the cassette
/*      reader via a parallel interface (8255 24-bit parrallel i/o chip).
/*      The data is stored in one of three 2k byte buffers and
/*      reformatted into an output file in CARP format.  The output
/*      file can be written to either virtual disk or hard disk.  Once
/*      on disk, the files can be copied to network, tape, floppy, etc.
/*
/*      The function which this program provides was originally written
/*      for an HP computer and later re-written for the LSI-11.  This
/*      re-write provides some modifications:
/*              1. Some Seadata readers have been modified to work only
/*                 with an LSI-11.  This modification was done to allow
/*                 the LSI-11 to latch data into its parallel port.  The
/*                 8255 chip used in this version is fast enough to
/*                 accept data from either version of the Seadata reader.
/*              2. There are two basic formats of input from Seadata
/*                 readers.  One is the CARP format and the other is
/*                 the Seadata format with data bytes and message bytes.
/*                 As with previous versions of the CARP program, this
/*                 version reads only the carp format.
/*              3. The output buffers are much larger than before.  The
/*                 default output buffer size is 8k bytes.  If smaller
/*                 buffers are needed, the OUTB parameter in seadata.h
/*                 will need to be modified and the programs recompiled
/*                 and linked.  Since the data is being written to disk,
/*                 there may be some loss of performance and timing
/*                 problems with smaller data buffers.
/*              4. The LSI-11 version wrote characters indicating
/*                 data read errors to the screen and/or printer.  This
/*                 version notes the type of errors on the screen and
/*                 prints a summary at the end of reading a tape.
/*              5. There is no output to a printer during reading.  A
/*                 file which summarizes the reading is written to disk
/*                 and can be printed after the reading is finished.
```



```
/*      This is the main program which controls the processing of the
/*      Seadata cassettes.
/*      This routine checks for the model number of the reader,
/*      gets the number of characters/record, opens the output file,
/*      and generally gets things set to go.
  */

        BUFF    D_buff;         /* Data buffers and pointers */
        D_Head  Datahead;       /* Headers output data buffers. */
        C_Head  Comhead;        /* Comment headers.  */
        C_field Comment;        /* Comment array.  */
        R_Head  Rechead;        /* Header. */

main ()
{
        time_t  tsec;                   /* type for time variable. */
        int     files(int *, int *, char *);
        int     handle (int);           /* collect data. */

        void    intr_init(void);        /* initialize interrupts. */
        void    seainit (void);         /* initialize data pointers */
        void    windows(void);          /* initialize the screen */
        void    e_line(void), p_line(void), m_line(void);
        void    read_com (int, char *);
        void    getrtime (int *), setdtime (int *);

        int     seacom(int);
        long    disk_free (char *);

        struct  text_info       text;   /* store screen info. */

        char    *tstring;       /* string for the time variable. */
        char    model[6], dr;

        unsigned long bytes;
        unsigned char i_mask;
        int     dtm[6];
        int     istat, jj, i, jstat;
        int     fh_d, fh_c;     /* File handles.*/

        char    com[72];        /* comment records */

        static char reader[]={"Model #"};
```

```
static char err1[]={"Input error  --  "};
static char err2[]={"the model type must be specified"};
static char err3[]=
        {"as either 0, 12, or 850.  Please reenter."};
static char err4[]=
        {"the model type must be 1-3 characters and specified"};
static char err5[]=
{"Unable to write comment information - continue <yes or no>? "};

static char m0[]={"0"};          /* String with the model types */
static char m12[]={"12"};        /* used for comparrison with */
static char m850[]={"850"};      /* what user enters. */

/*          Get the screen attributes and the time variables. */

gettextinfo (&text);
tzset();                         /* set the timezone variables */
time (&tsec);            /* get the current number of seconds */

tstring = ctime (&tsec);         /* get the time string */
i = strlen (tstring) -1;
tstring[i] = '\0';       /* remove newline at end */

/*          Set up pointer and buffer constants. */
D_buff.R_count = RAWB;           /* Usable space in input buffer */

D_buff.r_point[0] = &D_buff.Raw1[0];    /* Raw input pointers. */
D_buff.r_point[1] = &D_buff.Raw2[0];
D_buff.r_point[2] = &D_buff.Raw3[0];

D_buff.o_point[0] = &D_buff.Out1[0];    /* output pointers. */
D_buff.o_point[1] = &D_buff.Out2[0];
```



```
/*          Set up the window attributes. */

D_buff.topw = YELLOW + (BLUE << 4);
D_buff.p_attr = LIGHTGRAY + (BLACK << 4);
D_buff.m_attr = MAGENTA + (BLACK << 4);
D_buff.err_attr = RED + (GREEN << 4);
D_buff.e_attr = CYAN + (BROWN << 4);

window (1, 22, 80, 25);
textattr (D_buff.e_attr);
clrscr ();

window (1, 1, 80, 3);
textattr (D_buff.topw);
clrscr ();
gotoxy (1, 1);
cprintf ("PC/CARP - %4.2f  --  Time: ", VERSION);
gotoxy (27,1);
cprintf ("%s", tstring);

window (1, 4, 80, 21);
textattr (D_buff.p_attr);
clrscr ();
gotoxy (1,2);
cputs
("Unlike other versions of CARP, this program will always produce");
gotoxy (1,3);
cputs
("two disk files  --  one containing the data in CARP format and");
gotoxy (1,4);
cputs
("the other containing comments and header information.  The comment");
gotoxy (1,5);
cputs ("file is printable.");

gotoxy (1,7);
cputs
("Please enter the type of reader you are using.  Valid types are:");
gotoxy (1,8);
cputs ("Model 0 -");
gotoxy (20,8);
cputs ("0");
gotoxy (1,9);
cputs ("Model 12 -");
gotoxy (19,9);
cputs ("12");
gotoxy (1,10);
cputs ("Cartridge 850 -");
gotoxy (18,10);
cputs ("850");
```

```
/*              Loop to prompt the user and test the string returned
/*              for the cassette/cartridge reader type.
  */
        istat = 0;
        jj = 0;
        while (istat != 1)
        { window (1, 4, 80, 21);
          textattr (D_buff.p_attr);
          gotoxy (1, 12);
          clreol ();
          cprintf ("%s: ", reader);

/*              Get the model number.  If == NULL, then bad.
/*              print error and try again.
  */
          model[0] = 4;
          cgets (model);
          i = model[1];
          if (i == 0)
            { jj++;
              if (jj > 3)    goto quit;
              D_buff.er_line = 4;
              e_line ();
              cprintf ("%s%s", err1, err2);
              e_line ();
              cputs (err3);
            }
/*              If the length of the model string is < 4, this may be
/*              correct.  Check and then set the indicators for the
/*              reader type.
  */
          else if (i < 4)
            { if (strncmp (&model[2], m12, 1) == 0)
                { D_buff.irtyp = -1;            /* model 12 */
                  D_buff.adex = 1;
                  istat = 1; }
              else if (strncmp (&model[2], m0, 1)  == 0)
                { D_buff.irtyp = 0;             /* model 0 */
                  D_buff.adex = 2;
                  istat = 1; }

              else if (strncmp (&model[2], m850, 1) == 0)
                { D_buff.irtyp = 1;             /* 850 cartridge */
                  D_buff.adex = 0;
                  istat = 1; }
```



```
              else
                { jj++;
                  if (jj > 3)      goto quit;
                  D_buff.er_line = 0;
                  e_line ();
                  cprintf ("%s%s", err1, err2);
                  e_line ();
                  cputs (err3); }
            }
          else
            { jj++;
              if (jj > 3)      goto quit;
              D_buff.er_line = 0;
              e_line ();
              cprintf ("%s%s", err1, err4);
              e_line ();
              cputs (err3); }
        }

        windows ();
        window (1, 1, 80, 3);
        textattr (D_buff.topw);
        gotoxy (60,1);
        cprintf ("%s is %s", reader, &model[2]);
        D_buff.mes_line = 2;

/*      Start loop to read the data tapes.
  */
        jstat = 1;
        while (jstat == 1)

/*              Call the files routine to open the data file and
/*              get the file pointer initialized.
  */
        { if (files (&fh_d, &fh_c, &dr) != 1)  /* Return file handle. */
            { e_line();
              cputs ("Fatal file open failure  --  exiting!");
              goto quit;
            }
/*              Call the routine to get comments.  Load the header
/*              for the comments to the data file.
  */
          istat = seacom (fh_c);
          if (istat == -3)
            { D_buff.er_line = 0;               /* write error */
              e_line ();
              clrscr ();
              cputs (err5);
              jstat = yno ();
              continue; }
```

```
        else  if (istat == -5)
           {  D_buff.er_line = 0;            /* read error */
              e_line ();
              clrscr ();
              cputs
          ("Unable to read comment information - continue <yes or no>? ");
              jstat = yno ();
              continue;
           }

/*            Get the cassette reader ready and then have the user
/*            start the reading.  All of the reading will be done
/*            in a separate routine.
  */

        sealnit ();          /* init. the pointers, etc. */

/*            Calculate the number of bytes to add to make the
/*            output an even number of words.
  */
        i = (D_buff.r_c_count + (D_buff.r_c_count % 2)) / 2;
        D_buff.pad_count = (i + (i % 2)) - i;

        intr_init ();
        istat = handle (fh_d);
        if (istat != 0)
           {  p_line ();
              clrscr ();
              cprintf
             ("Fatal tape read error  --  status = %d", istat);
              p_line ();
              cputs ("Exiting  --");
              goto quit;
           }

/*            Print a summary of the tape reading.  */
        window (1, 18, 80, 21);
        textattr (D_buff.err_attr);

        gotoxy (19, 1);
        cprintf ("%6lu", D_buff.records);
        gotoxy (1, 4);
        cprintf ("   %5lu      %5lu          %5lu          %5lu",
                 D_buff.p_error, D_buff.l_error, D_buff.s_error,
                 D_buff.e_error);
```



```
/*            Read any comments for the end of the tape.
/*            Then, write the comments to the comment file.
  */
     read_com (1, tstring);
     for (i = 0; i < Comment.C_num; i++)
        {  strcpy (com, Comment.C_array[i]);       /* copy */
           for (jj = strlen(com); jj < 70; jj++)   /* pad  */
                com[jj] = ' ';
           com[70] = '\n';                         /* newline */
           istat = write (fh_c, com, 71);          /* write */
           if (istat != 71)
              {  D_buff.er_line = 0;       /* write error */
                 e_line ();
                 cputs (err5);
                 jstat = yno ();
                 continue; }
        }

/*            Update the time string and reset the time on the
/*            top window before going any further.
  */
     time (&tsec);            /* get current number of seconds */
     tstring = ctime (&tsec);         /* get the time string */
     i = strlen (tstring) -1;
     tstring[i] = '\0';       /* remove newline at end */

     window (1, 1, 80, 3);
     textattr (D_buff.topw);
     gotoxy (27, 1);
     cprintf ("%s     ", tstring);

/*            Check to see if the user wants to read more tapes */

     close (fh_c);
     close (fh_d);                    /* clean up files */
     D_buff.prm_line = 0;
     p_line ();
     clrscr ();
     bytes = disk_free (&dr);  /* Get free disk space. */
     cprintf ("%ld bytes free on drive %c:", bytes, dr);
     p_line ();
     cputs ("Would you like to read another tape <yes or no>? ");
     jstat = yno ();
  }
```

```
/*              Quit the program - reset the screen before exiting.
  */
quit:
        close (fn_c);
        close (fn_d);
        window (1, 1, 80, 25);
        textattr (text.normattr);
        gotoxy (1,25);

/*              Reset the real-time and DOS clocks as necessary.
/*
/*              Enable the real-time clock interrupt and disable
/*              the COM2 interrupt.
  */
        i_mask = inportb (0x21) | 0x08;         /* reset COM2  */
        outportb (0x21, (i_mask & 0xFE));       /* enable IRQ 0 */

/*              Get the value in the real-time clock and reset
/*              the DOS clock.
  */
        getrtime (dtm);         /* get real-time clock value */
        setdtime (dtm);         /* set DOS clock */

}
```

```
#include <conio.h>
#include <stdio.h>
#include <dos.h>
#include "seadata.h"

/*      seainit.c
/*      T.W. Danforth           15-Feb-1990
/*      Routine shich is used to initialize variables and pointers for
/*      the Seadata cassette reading program.
/*
/*      Parameters:
/*              none.
/*      Return values:
/*              none.
  */

void    seainit ()

{
        extern  BUFF    D_buff;         /* Data buffers and pointers */
        extern  D_Head  Datahead;       /* Header area data buffers. */
        extern  C_Head  Comhead;        /* Comment headers. */
        extern  C_field Comment;        /* Comment array. */
        extern  R_Head  Rechead;        /* Headers. */

        static char c1[]={"Cassette records:"};
        static char c2[]={"Processing errors:"};
        static char c3[]=
           {"  Parity   long records   short records   tape errors"};

/*              Set up the tape read error window. */

        window (1, 18, 80, 21);
        textattr (D_buff.err_attr);
        clrscr ();

        gotoxy (1,1);
        cputs (c1);
        gotoxy (1,2);
        cputs (c2);
        gotoxy (1,3);
        cputs (c3);
```



```
/*              Initialize data pointers and counters. */

        Rechead.R_head.errind = 0;      /* Clear error count. */
        Rechead.R_head.cas_no = 0;      /* First cassette record. */
        D_buff.records = 0;             /* Count - cassette records */

        Datahead.header.nrec = 1;       /* Output buffer number. */
        Datahead.header.errind = 0;     /* No errors during output. */
        Datahead.header.ncas = 0;       /* Cassette records in output */

        D_buff.s_error = 0;             /* Error counts */
        D_buff.l_error = 0;
        D_buff.p_error = 0;
        D_buff.e_error = 0;
        D_buff.m_error = 0;

        D_buff.er_line = 0;
        D_buff.mes_line = 0;
        D_buff.prm_line = 0;

        D_buff.which_raw = 0;           /* Indicate which buffers are */
        D_buff.which_out = 0;           /* being filled and emptied.  */
        D_buff.which_proc = 0;

        D_buff.parity.storage = 0;      /* Clear the parity for start */


               /* Set up start and end of raw & proc buffers. */
        D_buff.Rawpoint  = D_buff.r_point[D_buff.which_raw];
        D_buff.Raw_end   = D_buff.Rawpoint + D_buff.R_count *
               sizeof (char);
        D_buff.Procpoint = D_buff.r_point[D_buff.which_proc];
        D_buff.Proc_end  = D_buff.Procpoint + D_buff.R_count *
               sizeof (char);

               /* Point to starting location of Record header. */
        D_buff.rec_p = D_buff.o_point[0] + (10 * sizeof(short));
               /* Starting location for filling output buff. */
        D_buff.Outpoint = D_buff.rec_p + (3 * sizeof(short));
        D_buff.Out_end = D_buff.o_point[0] + OUTB;

        return;
}
```

```
#include "seadata.h"

/*      subs.c
/*      Subroutines used in the Seadata cassette reading programs.
/*      This is a collection of programs to calculate parity for a
/*      Seadata record and process 1, 2, or 3 characters in a
/*      Seadata transmission.
/*
/*      parity
/*      Routine to use an XOR operation on the Seadata cassette records
/*      to check the operation of the 4 channels in the recorder.
/*
/*      Parameters:
/*              value   - The word to be "added" to the XOR.
/*              parity  - Address where output parity is to be stored.
/*              flag    - Flag used to indicate the end of record.
/*                        = 0 , not the end.
/*                        = 1 , indicates the last word, finish
/*                              the parity calculation.
/*      Return values:
/*              = 0 if parity checks ok.
/*              = 1 if there is a parity error.
  */

int     parity (short int value, short int *parity, int flag)
{
        extern  BUFF    D_buff;         /* Data buffers. */
        unsigned char temp, temp1;      /* temporary byte storage. */

        D_buff.parity.storage = D_buff.parity.storage ^ value;
        if (flag == 0)
                return (0);             /* Not the end, return. */

/*              The end of the record has been found, need to finish
/*              XOR process by doing an XOR on each byte and then on
/*              upper and lower nibble in each byte.
  */
        temp = D_buff.parity.ps.s1 ^ D_buff.parity.ps.s2;
        temp1 = temp;
        temp = (temp >> 4) & 0x0F;      /* Get high 4 bits only */
        temp1 = temp1 & 0x0F;   /* Get the low 4 bits only  */

        *parity = temp ^ temp1; /* Produce final parity  --  */
                                /* should = 0x000F for odd parity. */
        if (*parity == 0x000F)
            return (0);         /* Parity is ok   */
        else
            return (1);         /* Parity is bad  */
}
```



```
/*      char_1
/*      Routine to perform necessary end of record processing for
/*      the last remaining character in the cassette record.
/*      This should only be the parity character.  Clear extra bits
/*      and finish the parity check.
/*
/*      parameters:
/*              value   - The word which contains the character to
/*                        be moved to the output buffer.
/*      Return values:
/*              none
  */

void    char_1 (short int value)
{
        extern  BUFF    D_buff;         /* Data buffers and pointers */

        union   {
                struct  {               /* bytes */
                        unsigned char s1;       /* high byte */
                        unsigned char s2;       /* low byte */
                        } b;
                short int t;            /* short word */
                } tmp;

        tmp.t = value & 0xF000;         /* Clear the 12 bits. */
        D_buff.p_stat = parity (tmp.t, &D_buff.p_value, D_buff.p_flag);

/*              Check move_flag.  If move_flag = 0, move last character
/*              onto a byte boundary.  If move_flag = 1, move last
/*              character onto a 1/2 byte boundary.
  */
        if (D_buff.move_flag == 0)
           {  tmp.b.s2 = tmp.b.s2 & 0xF0;       /* high bits only used */
              *(D_buff.Outpoint) = tmp.b.s2;
           }
        else
           {  tmp.b.s2 = (tmp.b.s2 >> 4) & 0x0f;  /* shift high bits */
                         /* to low and clear garbage bits shifted in */
              *(D_buff.Outpoint) = *(D_buff.Outpoint) | tmp.b.s2;
           }
        D_buff.Outpoint++;
        return ;
}
```

```
/*      char_2
/*      Routine to process two characters from the data word.
/*      Usually run at the end of the Cassette record to clean-up
/*      any extra bytes.  In this case, process the parity on two
/*      characters but move only one since the last should be the
/*      parity character.
/*
/*      parameters:
/*              value   - The data word which is to be processed.
/*      return values:
/*              none.
  */

void    char_2 (short int value)
{
        extern  BUFF    D_buff;         /* data buffers and pointers */

        union   {
                struct  {               /* bytes */
                        unsigned char s1;       /* high byte */
                        unsigned char s2;       /* low byte */
                        } b;
                short int t;            /* short word */
                } tmp;

        tmp.t = value & 0xFF00;         /* Clear the low byte */
        D_buff.p_stat = parity (tmp.t, &D_buff.p_value, D_buff.p_flag);

/*              Check move_flag.  If move_flag = 0, move last two
/*              characters onto a byte boundary.  If move_flag = 1, move
/*              last two characters onto a 1/2 byte boundary.
  */
        if (D_buff.move_flag == 0)
                                        /*      byte boundary  */
           { *(D_buff.Outpoint) = tmp.b.s2;   /* uppper byte */
             D_buff.Outpoint++; }
        else
                                /*      1/2 byte boundary.  */
           { tmp.t = (tmp.t >> 4) & 0x0ff0;     /* shift into position */
                        /* and clear any bits shifted in on left.*/
             *(D_buff.Outpoint) = *(D_buff.Outpoint) | tmp.b.s2;
             D_buff.Outpoint++;
             *(D_buff.Outpoint) = tmp.b.s1;
             D_buff.Outpoint++; }
        return;
   }
```



```
/*      char_3
/*      Routine to process 3 characters from the data word.  This will
/*      usually be used during the processing of the cassette record
/*      as well as at the end of the record.  The processing during the
/*      record involves just moving the characters.  At the end of the
/*      record, the LCC (parity) character is not moved.
/*
/*      parameters:
/*              value   - The data word which is to be processed.
/*              eor     - End-of-record flag.
/*                        = 0  - not end-or-record.
/*                        = 1  - end-of-record.
/*
/*      return values:
/*              none.
  */

void    char_3 (short int value, int eor)
{
        extern  BUFF    D_buff;

        union   {
                struct  {                       /* bytes in data word */
                        unsigned char s1;       /* high byte */
                        unsigned char s2;       /* low byte */
                        } b;
                short   int t;                  /* data word */
                } tmp;

        tmp.t = value & 0xFFF0;     /* Clear low 4 bits (flag bits) */
        D_buff.p_stat = parity (tmp.t, &D_buff.p_value, D_buff.p_flag);

/*
/*              Move all 3 characters.
/*              The move will depend on move_flag,
/*                      = 0  --  move onto a byte boundary.
/*                      = 1  --  move onto a 1/2 byte boundary.
  */

        if (D_buff.move_flag == 0)      /*      byte boundary  */
           { *(D_buff.Outpoint) = tmp.b.s2;   /* upper byte */
             D_buff.Outpoint++;
             *(D_buff.Outpoint) = tmp.b.s1;     /* lower byte  */
             D_buff.move_flag = 1;
             if (eor == 1)
                D_buff.Outpoint++;
           }
        else
```

```
                                        /*      1/2 byte boundary.  */
        { tmp.t = (tmp.t >> 4) & 0x0fff;      /* shift into position */
                    /* and clear any bits shifted in on left */
          *(D_buff.Outpoint) = *(D_buff.Outpoint) | tmp.b.s2;
          D_buff.Outpoint++;
          *(D_buff.Outpoint) = tmp.b.s1;
          D_buff.Outpoint++;
          D_buff.move_flag = 0;
        }
      return;
}
```

```
/*      time.c
/*      K. Prada
/*      Routines which get or set the real-time clock or the DOS
/*      clock.  These routines use the software interrupts in
/*      bios to do the set and reset.
/*      All parameters are passed to and from the interrupt routines
/*      through the registers defined in REGS in dos.h.
  */

#include <dos.h>
union REGS timsr,timrr;

void    getrtime (int *dtm)     /* get AT real time clock */
{

/*      Get the time and date from the real time clock.
/*      Parameters:
/*              *dtm    - Pointer to integer array which will contain the
/*                        time and date as follows:
/*                        dtm[0] = year
/*                        dtm[1] = month
/*                        dtm[2] = day
/*                        dtm[3] = hours
/*                        dtm[4] = minutes
/*                        dtm[5] = seconds
/*
/*      Returns:
/*              none.
  */

/*              Get real time clock thru bios     */

        timsr.h.ah = 0x02;
        int86 (0x1a, &timsr, &timrr);
        dtm[5] = bcdbin (timrr.h.dh);           /* return s
        dtm[4] = bcdbin (timrr.h.cl);           /* return
        dtm[3] = bcdbin (timrr.h.ch);           /* return

/*              Get the real-time clock date   */

        timsr.h.ah = 0x04;
        int86 (0x1a, &timsr, &timrr);
        dtm[2] = bcdbin (timrr.h.dl);           /* return day */
        dtm[1] = bcdbin (timrr.h.dh);           /* return month */
        dtm[0] = bcdbin (timrr.h.cl);           /* return year */
}
```



```
int     bcdbin (int val)
{
/*              Routine to convert BCD (binary-coded decimal?) values
/*              to decimal values.
  */
        return ((((val & 0x00f0) >> 4) * 10) + (val & 0x000f));
}

void    setrtime (int *dtm)     /* set AT real time clock */
{
/*      setrtime
/*      Routine to set tha AT's realtime clock.
/*      Parameters:
/*              *dtm    - For explanation, see getrtime.
/*
/*      Returns:
/*              none.
  */

/*              Set real time clock thru bios    */

        timsr.h.dh = binbcd (dtm[5]);           /* seconds */
        timsr.h.cl = binbcd (dtm[4]);           /* minutes */
        timsr.h.ch = binbcd (dtm[3]);           /* hours */
        timsr.h.ah = 0x03;
        int86 (0x1a, &timsr, &timrr);

/*              Set the real-time clock date     */

        timsr.h.dl = binbcd (dtm[2]);           /* day */
        timsr.h.dh = binbcd (dtm[1]);           /* month */
        timsr.h.cl = binbcd (dtm[0]);           /* year */
        timsr.h.ch = 0x19;                      /* century */
        timsr.h.ah = 0x05;
        int86 (0x1a, &timsr, &timrr);
}


int     binbcd (int val)
{
/*      binbcd
/*      Routine to convert binary to BCD values for the time routine.
  */
        int v;

        v = val/10;
        return ((v << 4) + (val - (v * 10)));
}
```

```
void    getdtime (int *dtm)     /* get DOS time */
{
/*      getdtime
/*      Routine to get the DOS system time.
/*      Parameters:
/*              *dtm    - Pointer to integer array - see getrtime.
/*      Returns:
 *              none.
 */
/*              Get dos system time     */

        timsr.h.ah = 0x2c;
        int86 (0x21, &timsr, &timrr);
        dtm[5] = timrr.h.dh;                    /* return seconds */
        dtm[4] = timrr.h.cl;                    /* return minutes */
        dtm[3] = timrr.h.ch;                    /* return hours */

/*              Get dos system date     */

        timsr.h.ah = 0x2a;
        int86 (0x21, &timsr, &timrr);
        dtm[2] = timrr.h.dl;                    /* return day */
        dtm[1] = timrr.h.dh;                    /* return month */
        dtm[0] = timrr.x.cx - 1900;             /* return year */
}

void    setdtime (int *dtm)     /* set DOS time */
{
/*      setdtime
/*      Routine to set the DOS system time.
/*      Parameters:
/*              *dtm    - See getrtime for explanation.
/*      Returns:
/*              none.
 */
/*              Set the DOS system time         */

        timsr.h.dh = dtm[5];
        timsr.h.cl = dtm[4];
        timsr.h.ch = dtm[3];
        timsr.h.ah = 0x2d;
        int86 (0x21, &timsr, &timrr);

/*              Set the DOS system date         */

        timsr.h.dl = dtm[2];
        timsr.h.dh = dtm[1];
        timsr.x.cx = dtm[0] + 1900;
        timsr.h.ah = 0x2b;
        int86 (0x21, &timsr, &timrr);
}
```

```
#include <conio.h>
#include "seadata.h"

/*      windows.c
/*      T.W. Danforth           20-Feb-1990
/*      Routine to set up and clear windows for the
/*      carp tape reading program.
/*
/*      Parameters:
/*              none.
/*      Returns:
/*              none.
  */

void    windows (void)
{
        extern  BUFF    D_buff;         /* Data buffers. */

        window (1, 4, 80, 11);
        textattr (D_buff.p_attr);
        clrscr ();

        window (1, 12, 80, 17);
        textattr (D_buff.m_attr);
        clrscr ();

        window (1, 22, 80, 25);
        textattr (D_buff.e_attr);
        clrscr ();

        window (1, 18, 80, 21);
        textattr (D_buff.err_attr);
        clrscr ();

        return;
}
```



```
#include <conio.h>
#include <stdio.h>
#include <string.h>

/*      yno.c
/*      T.W. Danforth           26-aug-89
/*      Integer function to read a response from stdin and check
/*      to see if it is either a yes or no.  The value returned from
/*      this function is
/*              0 = error
/*              1 = yes
/*              2 = no
  */

int     yno (void)
{
        char    ans[6], *c = &ans[2];
        register int  len;

        ans[0] = 4;             /* limit response length */

/*      read the response and convert it to lower case */
        cgets (ans);

        strlwr (c);

/*              Compare the lowercase response to "yes" and "no" to see
/*              if they are true.  First check the length of the
/*              response string if it is 0 or > 3 characters,
/*              error  --  return 0.
  */
        len = ans[1];

        if (len == 0 || len > 3)
                return (0);

/*              Compare with both "yes" and "no" to see if correct. */

        if (strncmp (c, "yes", len) == 0)
           return (1);
        else
           {  if (strncmp (c, "no", len) == 0)
              return (2);
           }

/*      if all else fails, return a failure. */

        return (0);
}
```

Appendix 6: Bibliography

Hunt, Mary. CARP Program documentation. Woods Hole Oceanographic Institution, Information Processing Center, August, 1972.

Model 12B 4-Track Cassette Tape reader, User Operation Manual; Sea Data Corporation, Newton, Massachusetts, December, 1982.

In addition to the above references, the following items were used during the writing of these programs and are listed here for completeness.

Biggerstaff, Ted J. Systems Software Tools. Prentice-Hall, Englewood Cliffs, New Jersey ,1986; pp 69–86.

Duncan, Ray; Advanced MSDOS Programming; Microsoft Press, Redmond, Washington, 1988.

Jourdain, Robert; Programmer's Problem Solver for the IBM PC, XT, and AT; Brady Books, New York, New York, 1986; pp. 17–25.

LAN WorkPlace Network Software, User's Reference Manual. Excelan, Inc., San Jose, California, 1989.

Technical Reference, Personal Computer Hardware Reference Library; International Business Machines Corporation, Boca Raton, Florida, 1984, reference number 1502494; Chapter 1, System Board.

Turbo C User's Guide, Version 2.0; Borland International, Scotts Valley, California, 1988.

Turbo C Reference Guide, Version 2.0; Borland International, Scotts Valley, California, 1988.

DOCUMENT LIBRARY

January 17, 1990

Distribution List for Technical Report Exchange

Attn: Stella Sanchez-Wade
Documents Section
Scripps Institution of Oceanography
Library, Mail Code C-075C
La Jolla, CA 92093

Hancock Library of Biology &
Oceanography
Alan Hancock Laboratory
University of Southern California
University Park
Los Angeles, CA 90089-0371

Gifts & Exchanges
Library
Bedford Institute of Oceanography
P.O. Box 1006
Dartmouth, NS, B2Y 4A2, CANADA

Office of the International
Ice Patrol
c/o Coast Guard R & D Center
Avery Point
Groton, CT 06340

NOAA/EDIS Miami Library Center
4301 Rickenbacker Causeway
Miami, FL 33149

Library
Skidaway Institute of Oceanography
P.O. Box 13687
Savannah, GA 31416

Institute of Geophysics
University of Hawaii
Library Room 252
2525 Correa Road
Honolulu, HI 96822

Marine Resources Information Center
Building E38-320
MIT
Cambridge, MA 02139

Library
Lamont-Doherty Geological
Observatory
Colombia University
Palisades, NY 10964

Library
Serials Department
Oregon State University
Corvallis, OR 97331

Pell Marine Science Library
University of Rhode Island
Narragansett Bay Campus
Narragansett, RI 02882

Working Collection
Texas A&M University
Dept. of Oceanography
College Station, TX 77843

Library
Virginia Institute of Marine Science
Gloucester Point, VA 23062

Fisheries-Oceanography Library
151 Oceanography Teaching Bldg.
University of Washington
Seattle, WA 98195

Library
R.S.M.A.S.
University of Miami
4600 Rickenbacker Causeway
Miami, FL 33149

Maury Oceanographic Library
Naval Oceanographic Office
Bay St. Louis
NSTL, MS 39522-5001

Marine Sciences Collection
Mayaguez Campus Library
University of Puerto Rico
Mayagues, Puerto Rico 00708

Library
Institute of Oceanographic Sciences
Deacon Laboratory
Wormley, Godalming
Surrey GU8 5UB
UNITED KINGDOM

The Librarian
CSIRO Marine Laboratories
G.P.O. Box 1538
Hobart, Tasmania
AUSTRALIA 7001

Library
Proudman Oceanographic Laboratory
Bidston Observatory
Birkenhead
Merseyside L43 7 RA
UNITED KINGDOM

50272-101

REPORT DOCUMENTATION PAGE	1. REPORT NO. WHOI-90-44	2.	3. Recipient's Accession No.

4. Title and Subtitle
The Seadata Program

5. Report Date
October, 1990

6.

7. Author(s)
Thomas W. Danforth

8. Performing Organization Rept. No.
WHOI-90-44

9. Performing Organization Name and Address
Woods Hole Oceanographic Institution
Woods Hole, Massachusetts 02543

10. Project/Task/Work Unit No.

11. Contract(C) or Grant(G) No.
(C) N00014-84-C-0134
(G) 14-08-0001-A0245

12. Sponsoring Organization Name and Address
Office of Naval Research

13. Type of Report & Period Covered
Technical Report

14.

15. Supplementary Notes
This report should be cited as: Woods Hole Oceanog. Inst. Tech. Rept., WHOI-90-44.

16. Abstract (Limit: 200 words)

Current meter and meteorological instrument data are typically stored in the instrument on cassette tapes. Seadata, described in this report, is a PC version of the original CARP program (CAssette Reading Program) which transferred the data and prepared it for further processing. Also described are two programs which provide byte swapping which is necessary to use the PC data on a VAX/VMS computer. Some changes to the CARP format have been made and are documented here.

17. Document Analysis a. Descriptors
sea data cassette reading program
CARP - PC version
seadata - PC version of CARP

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Approved for public release; distribution unlimited.		77
	20. Security Class (This Page)	22. Price